Scene understanding through Deep Learning

Luis Octavio Arriaga Camargo

Publisher: Dean Prof. Dr. Wolfgang Heiden

University of Applied Sciences Bonn-Rhein-Sieg,
Department of Computer Science

Sankt Augustin, Germany

May 2017

Technical Report 02-2017

Hochschule
Bonn-Rhein-Sieg
University of Applied Sciences

ISSN 1869-5272

ISBN 978-3-96043-045-2

Digital Object Identifier doi:10.18418/978-3-96043-045-2

Research and Development Project

Scene understanding through Deep Learning

Luis Octavio Arriaga Camargo
octavio.arriaga@smail.inf.h-brs.de

B-IT Master Studies Autonomous Systems

University of Applied Sciences Bonn-Rhein-Sieg
Heriot Watt University

Advisors:
Prof. Dr. Paul G. Plöger
paul.ploeger@h-brs.de

MSc. Matías Valdenegro
m.valdenegro@hw.ac.uk

January 14, 2017

Abstract

Recent work in image captioning and scene-segmentation has shown significant results in the context of scene-understanding. However, most of these developments have not been extrapolated to research areas such as robotics. In this work we review the current state-of-the-art models, datasets and metrics in image captioning and scene-segmentation. We introduce an anomaly detection dataset for the purpose of robotic applications, and we present a deep learning architecture that describes and classifies anomalous situations. We report a METEOR score of 16.2 and a classification accuracy of 97 %.

Acknowledgements

I would like to thank everyone who has contributed to the creation of the anomaly detection dataset: Martha Camargo, Luis Arriaga, Catherine Capellen, Daniel Vázquez, José Mayoral, Anakaren Sepúlveda, Janssel Matías, Roberto Mendieta, Argentina Ortega, Gabriela Cortés, Roberto Cai, Ha Duc Bach, Alejandro Castro, Santosh Thoduka, Isadora Rodríguez, Anand Ajemera, Saikiran Kannaiah, Irynka Mandarynka and Lora Lyudmilova. I specifically thank Catherine Capellen for the revisions made to the text.

Finally, I would like to thank my parents Luis Arriaga Reyes and Martha Camargo Villa for their everlasting love and support.

Contents

1 Introduction

During the last five years the current number of robots used in industrial and service applications has increased by more than 25% [1]. Domestic and social robots are currently being employed in a wide range of applications such as, nurse assistants, warehouse patrols, household services and personal intelligent assistants. Therefore, modern robot applications require a high-level of human interaction along with the ability to interact in a set of very diverse circumstances; these requirements have led to an extensive amount of research on human-robot interaction, and general-purpose artificial intelligence [42]. Unfortunately, the current robot generation is unable to attend these demands since in most applications the robot's intelligence is developed for a very limited scope. We argue that in order to create the next successful generation of robot applications, it is important to investigate further on algorithms that widen the robot's ability to interact humanly and to understand human behaviour in more general situations. Making a machine understand and communicate using natural language is one of the hardest open problems in artificial intelligence (AI). However, a recently developed set of feature learning algorithms has been able to overcome long lasting problems in AI and machine learning (ML). These algorithms are called deep learning algorithms, and they have been successfully applied to the problem of understanding a scene; either through scene-segmentation or image captioning. Therefore, the research herein focuses on investigating the state-of-the-art algorithms for scene-understanding, while pondering their further development for robotic platforms. For this latter task, we propose a potential robot functionality, in which the agent has to classify and describe anomalous situations. Thus, our main objectives can be summarized as: create a comprehensible review and evaluation of the state-of-the-art algorithms for scene understanding, and ultimately, generate a new robot ability in the context of anomaly detection, such that it communicates potentially dangerous situations using natural language, a constrained amount of sensor data; and most importantly, by not relying on in any predefined hand-crafted model.

This work is divided into three sections: the first section contains the theoretical background and it consist of three parts, the first part gives an overall review on artificial neural networks and introduces the reader to the notation used in subsequent sections. The next part gives a rigorous description of convolutional neural networks (CNNs) since they form the basis of scene-understanding algorithms. The final part of this section reviews recurrent neural networks (RNNs); more specifically long short-term memory networks (LSTMs). The second section contains the review of the state-of-the art algorithms in scene understanding; more specifically, image captioning and image segmentation. Finally, the last section describes our neural network architecture used for captioning anomalous images, along

with the dataset created for this purpose, and the hardware and software considerations made to include our model in robot platforms.

2 Artificial neural networks

Artificial neural networks (ANNs) are biologically inspired architectures for approximating real-valued functions [34]. The main idea behind ANNs is that there exists connections between single-processing units that we call neurons; these connections are often referred to as weights and are essentially the free parameters of the model. Then, the output of a single neuron is non-linear scalar function applied to a linear combination of the incoming inputs to the neuron. This output can be later used by another set of neurons as its input. There exist many ANN architectures that account for different ways in which the connections between neurons are imposed; however, on this section we will only refers to feed-forward neural networks.

In order to approximate a function the weights if the ANN are changed by an optimization algorithm which minimizes the error between a target function and the function which is calculated by the ANN. One of the most used optimization algorithms is stochastic gradient descent (SGD) [34]. SGD has proven to be superior for ANNs since the cost function to minimize is usually a high-dimensional manifold, with a high number of local minima and saddle points [4]; consequently, calculating an approximated gradient often aids to escape these critical points [34].

2.1 Multi-layer perceptron architecture

One of the most successful ANN architectures for supervised learning is the multi-layer perceptron (MLP) [34]. The MLP is a feed-forward ANN, and its architecture consists of an input layer, one or several hidden layers and one output layer. The input layer is often a vector of features, and the output layer gives the approximated target values calculated by the network for a given input. Each of the hidden layers and the output layer, consists of several units called perceptrons; also named *neurons*. Each neuron in these layers is connected to several input values from previous layers, and it outputs a single value which is connected to neurons in the next layer. In a typical MLP architecture, a single neuron is connected too all values from the previous layer, and it sends its output to all neurons in the next layer; therefore, this architecture is also called fully connected network, or dense network. An MLP architecture with one hidden layer can be observed in figure 1. We will later see that restricting the network connections; and consequently having less parameters, will lead to a better performance on more specialized input values, such as images or time series [4]; however, for the remaining parts of this section, we will only describe fully-connected networks.

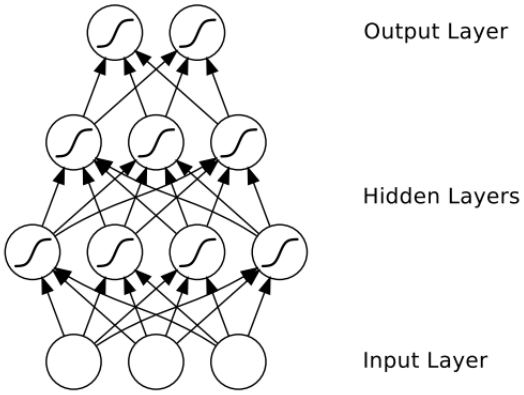


Figure 1: [21] MLP architecture

The operation made on each neuron consists on multiplying each input value from previous layers with a corresponding weight, summing up all of these values, and finally applying a non-linear differentiable function to this sum. Some of the most typical non-linear activation functions used in MLPs can be observed in figure 2. This non-linear differentiable function σ would remain undefined since it is chosen based on the application at hand; however, one of the most common functions used for σ is the point-wise sigmoid function

$$\sigma(x) = \frac{1}{1 + \exp^{-x}} \tag{1}$$

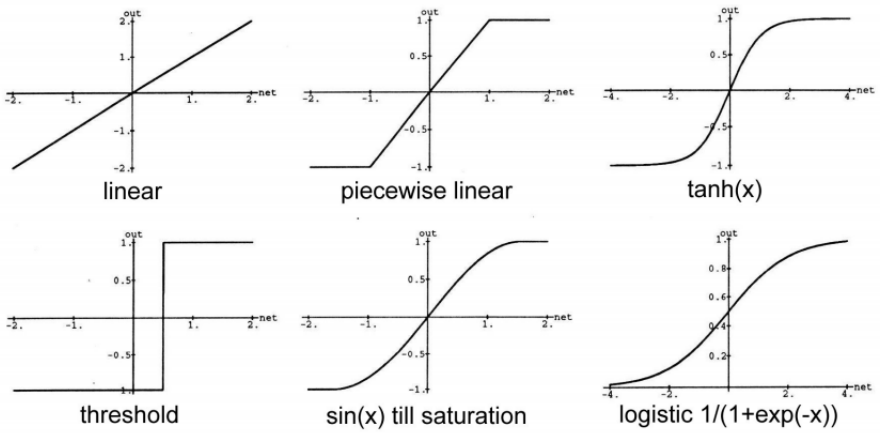


Figure 2: [21] Typical activation functions used in ANNs

Another common function used in modern architectures, is the point-wise rectified linear unit (ReLU) function.

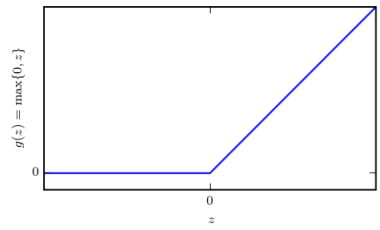


Figure 3: [4] ReLU activation function.

Even though this function is not differentiable at zero, it has proven in practice that this does not hurt the optimization algorithm, it promotes sparsity in the model, and that it has achieved better performance on several datasets when compared with other activation functions [20]. The ReLU equation can be observed below, as well as the plot of the function in figure 3

$$\sigma(x) = \max(0, x) \tag{2}$$

We will now define all the architecture values and connections more precisely. We refer to the input values of each neuron in layer l as x_i^{l-1}, note that the input values come from neurons in the previous layer; therefore, we refer to them as being in the layer $l-1$. Also, the index i runs for each neuron in that previous layer. The weights for each neuron j in layer l connected to the neuron i in the layer $l-1$ is defined as w_{ji}^l. Note that the weights are taken as being defined in the current layer l. The multiplication and then sum of the input values and its corresponding weights will be defined as net_j^l

$$net_j^l = \sum_i w_{ji}^l x_i^{l-1} \tag{3}$$

Finally, we will refer to the output of each neuron j in layer l as z_j^l. Therefore, every neuron j in the hidden layer l will perform the following operation:

$$z_j^l(w_{ji}^l) \equiv \sigma_j^l(net_j^l(w_{ji}^l)) = \sigma_j^l(\sum_i w_{ji}^l x_i^{l-1}) \tag{4}$$

One important remark is that the neuron also contain a *bias* term. This bias can be considered as a weight that does not depend on previous input

values; therefore, it can be naturally included in every weights first parameter, by defining that its corresponding input value will always be equal to one.

If we consider a MLP architecture of a single hidden layer, the network output would have following form

$$z_j^3 = \sigma_j^3(net_j^3) \tag{5}$$

Here the layer 3 correspond to the output layer, and consequently layer 2 is the hidden layer and layer 1 the input layer. Substituting the value of net_j^3 into the equation above, we obtain the following result

$$z_j^3 = \sigma_j^3(\sum_i w_{ji}^3 x_i^2) \tag{6}$$

From the equation above we can observe that the input values x_i^2 are actually the output values of that same layer z_i^2; therefore, we will then have the following

$$z_j^3 = \sigma_j^3(\sum_i w_{ji}^3 z_i^2) \tag{7}$$

Substituting the output values of the hidden layer

$$z_j^3 = \sigma_j^3(\sum_i w_{ji}^3 \sigma_i^2(\sum_m w_{im}^2 x_m^1)) \tag{8}$$

Here the values x_m^1 correspond the actual input values that come from the feature vector, and the values z_j^3 would be the output vector of the whole network. This last equation describes what we call forward propagation, since we propagate the values from the feature vector sequentially in the network until we reach the output layer. Two final remarks should be considered: first, we can generalize the network by including more hidden layers. This is easily done by substituting again the input values x_m^l, for the output values of the previous layer z_i^l, and continue substituting accordingly with its already defined operations. The second remark is that the output of the network has solely the weights as its free parameters. This weights are typically represented as knobs that can be turned to change the model's output. Therefore, the learning task consists of finding the set of weights that better approximate the target function, this step will be addressed on the next section.

2.2 Back-propagation algorithm for MLP

On the previous section we derived the forward propagation of a MLP architecture. This architecture has as free parameters a set of weights in every

layer. Now we will consider a solution to the problem of finding a set of weights that better approximate the target function. If we consider a supervised learning problem, we have a set of d input and target values, which we will denote as $\{(x_1^1, t_1), ..., (x_d^1, t_d)\}$ respectively. Naturally, the input values correspond to the input layer and take the superscripts of the first layer. Since we are considering a supervised learning problem, we should define a cost function that measures the error between the output of our network z_k^L and the target values (ground-truth values) t_k. A natural option is a variant of the mean squared error (MSE). Other cost functions are used depending on the application at hand; for example, a categorical cross-entropy is often used in classification problems. For the rest of this section we will continue to use the following MSE variant as the cost function; however, any generalization of the back-propagation algorithm can be easily obtained by substituting this cost function and its further derivatives.

Similarly to Mitchell [34] We define our cost function as

$$C = \frac{1}{2} \sum_{k \in outputs} (t_k - z_k^L)^2 \tag{9}$$

We would like to numerically minimize this equation with respect to all its weights. In other words we would like to minimize the error that we associated to the training examples with respect to all its independent variables. One of the most successful algorithms for ANNs for doing so, is the stochastic gradient descent (SGD) algorithm [34]. SGD takes the predictions of a single or multiple training examples, computes the cost function, calculates the gradient of this cost with respect to its weights and updates the weights using the following first order approximation [34]

$$w_{ji}^l \leftarrow w_{ji}^l + \Delta w_{ji}^l \tag{10}$$

In more detail, we update the previous weight with the term Δw_{ji}^l, which corresponds to an approximated direction of the steepest descent. We know that the steepest descent is obtained by calculating the negative gradient of the function; however, it is common practice to reduce the step taken, by multiplying the gradient with learning factor η.

$$\Delta w_{ji}^l = -\eta \nabla_{w_{ji}^l} C(w_{ji}) \tag{11}$$

Therefore, our task gets reduced to calculating the gradient with respect to all the weights w_{ji}^l in all the layers l. The SGD algorithm is able escape poor local minima, and has proven to converge to a good local minimum in many practical applications [34]. It is important to understand that we often don't wish to converge to the global minimum, since the global minimum is

likely to represent an over-fitted function of our specific training dataset [4].

We will start computing the gradient of the last layer weights w_{ji}^L. Note that we could have started at any layer, but going from the last one to the first one would prove to be efficient when calculating the derivatives. This efficiency is the main idea behind the back-propagation algorithm. The cost function dependency with respect to the set of weights in its last layer, can be written explicitly by composing the already defined functions of the network

$$C(w_{ji}^L) = C(z_j^L(\sigma_j^L(net_j^L(w_{ji}^L)))) \tag{12}$$

Now if we would like to calculate the gradient of the cost function with respect to these weights, we would simply apply the chain-rule to $C(w_{ji}^L)$.

$$\nabla_{w_{ji}^L} C(w_{ji}^L) = \frac{\partial C}{\partial z_j^L} \frac{\partial z_j^L}{\partial \sigma_j^L} \frac{\partial \sigma_j^L}{\partial net_j^L} \frac{\partial net_j^L}{\partial w_{ji}^L} \tag{13}$$

By calculating the first partial derivative, we obtain the following equation

$$\frac{\partial C}{\partial z_j^L} = \frac{\partial}{\partial z_j^L} \frac{1}{2} \sum_{k \in outputs} (t_k - z_k^L)^2 \tag{14}$$

Looking closely at the equation above, we do not have to expand the sum for every index j since all except one of those k values will be the actual variable for which we are taking the derivative of. More specifically, we can drop all indices for which $j \neq k$ since for all these cases, the derivative will be zero. Consequently, all the values left are those where $j = k$.

$$\frac{\partial C}{\partial z_j^L} = \frac{\partial}{\partial z_j^L} \frac{1}{2} \sum_j (t_j - z_j^L)^2 = -(t_j - z_j^L) \tag{15}$$

The second partial derivative is equal to one since the function $z_j^L = \sigma_j^L$. In a MLP architecture the z_j^L takes this trivial form, and one can think of it as naming differently the output of the activation function and the actual output of that layer.

The next partial derivative corresponds to the derivative of the activation function. Since this function definition depends on the corresponding application, we will define its derivative implicitly as:

$$\frac{\partial \sigma_j^L}{\partial net_j^L} = \frac{\partial}{\partial net_j^L} \sigma_j^L(net_j^L) = \sigma_j'^L \tag{16}$$

The last partial derivative gives us:

$$\frac{\partial net_j^L}{\partial w_{ij}^L} = \frac{\partial}{\partial w_{ij}^L} \sum_i w_{ji}^L x_i^{L-1} = x_i^{L-1} \tag{17}$$

Therefore the gradient of the cost function with respect to the last set of weights is equal to

$$\nabla_{w_{ji}^L} C = -(t_j - z_j^L)\sigma_j'^L x_i^{L-1} \tag{18}$$

Until now, the gradient was only derived with respect to the weights of the last layer; however, in order to optimize the weights of previous layers we would have to calculate the gradient with respect to these previous sets of weights. By previous we mean weights in layers $L-1, L-2, ..., 1$. To do so, we will follow the same procedure and start by defining explicitly the dependency of cost function with respect to this previous set of weights. We will see that in order to calculate this gradient, we can use the previously calculated derivatives; therefore, making the computation more efficient. This result eventually leads us to a recurrence relation that lets us calculate the derivative of any set of weights at any layer by only using an error associated to its upper layer [34].

The cost function dependency on the weights of a previous layer can be written as

$$C(w_{im}^{L-1}) = C(z_j^L(\sigma_j^L(net_j^L(z_i^{L-1}(\sigma_i^{L-1}(net_i^{L-1}(w_{im}^{L-1}))))))) \tag{19}$$

The explicit output of this network would then be:

$$C(w_{im}^{L-1}) = \frac{1}{2}\sum_j (t_j - z_j^L(\sigma_j^L(\sum_i w_{ji}^L \sigma_i^{L-1}(\sum_m w_{im}^{L-1} x_m^{L-2}))))^2 \tag{20}$$

Then the gradient with respect to the weights w_{im}^{L-1} would have the following form

$$\nabla_{w_{im}^{L-1}} C(w_{im}^{L-1}) = \frac{\partial C}{\partial z_j^L}\frac{\partial z_j^L}{\partial \sigma_j^L}\frac{\partial \sigma_j^L}{\partial net_j^L}\frac{\partial net_j^L}{\partial z_i^{L-1}}\frac{\partial z_i^{L-1}}{\partial \sigma_i^{L-1}}\frac{\partial \sigma_i^{L-1}}{\partial net_i^{L-1}}\frac{\partial net_i^{L-1}}{\partial w_{im}^{L-1}} \tag{21}$$

We can observe that the first three partial derivatives were already calculated when calculating the gradient for the weights w_{ji}^L. We can also conclude that if we want to optimize the weights on previous layers we will always have to calculate this first three derivatives; therefore, it is convenient to define them as the error associated with each neuron j on the last layer L [34].

$$\delta_j^L = \frac{\partial C}{\partial z_j^L}\frac{\partial z_j^L}{\partial \sigma_j^L}\frac{\partial \sigma_j^L}{\partial net_j^L} = -(t_j - z_j^L)\sigma_j'^L \tag{22}$$

We continue by calculating the fourth partial derivative

$$\frac{\partial net_j^L}{\partial z_i^L} = \frac{\partial}{\partial z_i^L} \sum_i w_{ji}^L z_i^L = w_{ji}^L \tag{23}$$

If we stop to think how the gradient is forming with these first four partial derivatives, we can observe that the gradient; and consequently the error that will get added to each weight, is the multiplication of the error associated to each output neuron (δ_j^L) times the weight that connects that neuron j to the previous unit i. This is shown on the figure 4

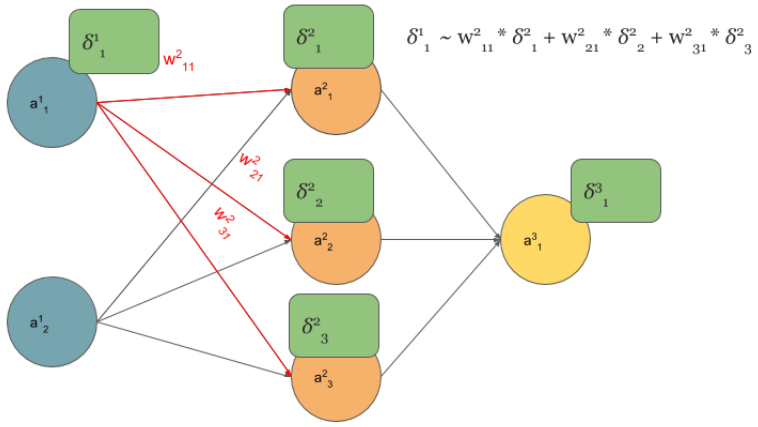


Figure 4: [23] Partial formation of the associated error given to a neuron

The last three derivatives are easily calculated and will result again on: a one, the derivative of the activation function, and the set of input values x_{mi} correspondingly. Putting together all the partial derivatives, we have the following equation for the gradient

$$\nabla_{w_{im}^{L-1}} C = (\sum_j \delta_j^L w_{ji}) \sigma_i'^{L-1} x_m^{L-2} \tag{24}$$

Now we are in position to generalize our results and calculate the gradient in respect to any set of weights at any layer, only by taking into consideration that the composition of functions continues as defined for a typical MLP architecture. Therefore, it would be convenient to also define an error to the hidden neurons. This error is different compared to the error of the output neurons δ_j^L, since the partial derivatives are also different. We then proceed to define δ_i^{L-1} as the error associated to the neurons located in the hidden layer:

$$\delta_i^{L-1} = (\sum_j \delta_j^L w_{ji}^L)\sigma_i'^{L-1} \tag{25}$$

We can generalize these results for all previous layers, by repeating this process and associating an error to each neuron similar to the one defined for the neurons i in the layer $L-1$ (δ_i^{L-1}) but now running on any layer l behind the output layer.

$$\delta_m^{l-1} = (\sum_n \delta_n^l w_{nm}^l)\sigma'^{l-1}[23] \tag{26}$$

Notice that we also dropped the summation index m on the σ'^{l-1}, since typical MLPs consider applying the same activation function for every neuron on the same layer; consequently, also its derivative is the same for every element m. Finally, the equation above can also be defined more generally by not providing the explicit calculation of the partial derivatives, but assuming some composition of functions between hidden layers:

$$\delta_m^{l-1} = \sum_n \delta_n^l \frac{\partial net_j^l}{\partial net_i^{l-1}} \tag{27}$$

Using the equations above, we can now calculate the complete gradient for any set of weights by calculating the neurons error times its input values of the layer below (its last partial derivative). We can repeat this process, and calculate each neuron's update until the input layer. This holds true, since the network architecture before the output layer is constituted of the same composition of functions as described for the error δ_i^{L-1}. This procedure of calculating efficiently the gradient of the cost function by starting from the last layer L weights, and then calculating for each layer behind it, is what we call the back-propagation algorithm [4].

3 Convolutional neural networks

Convolutional neural networks (CNNs/ConvNets) are artificial neural networks, that impose constraints on the weights in order to account for a more specialized input data [4]. These constraints lead to a convolution operation instead of the usual matrix multiplication used before the activation function. They are specialized to process data that has a grid-like topology, such as images or time-series. In a very general form, a CNN would receive this grid-like input data, and its task would be to learn a set of weights that get convolved in order to produce an output that contains a set of salient features. For example, if we take as input an image, the task of a CNN would be to learn kernel weights, in order to obtain features; such as, edges, circles, and eventually more abstract shapes.

As described in [49] [4] CNNs have four key ideas: local connectivity, parameter sharing, equivariant representation and the use of a hierarchical representation. By local connectivity we mean that the input of each neuron comes from a neighborhood of spatially located values; consequently, giving importance only to local connections. Parameter sharing; or sparse weights, means that the number of parameters that need to be stored is small in comparison to the input space. This property is also called tied weights, since each kernel is used at every position of the input; therefore, instead of learning a set of parameters for every location, they learn one set for all locations. This property translates in the case of CNNs applied to image processing, as being able to detect salient features using only tens of weight values in the kernels, while having input images of millions of pixels. Local connectivity and parameter sharing causes a layer to be equivariant to translations; therefore translations on the input data also give a translated output. Finally, the use of several convolutional layers with non-linear activations makes CNNs a deep learning method, since it builds abstract representations of the data from raw input values, in a hierachical manner.

One important remark about CNNs is that they are not invariant to transformations, such as translation, rotations, change of scale, or more generally to affine transformations. Consequently, other types of mechanisms are needed in order to handle such transformations and invariances of the input space. [25]

CNN-based models hold the sate-of-the-art performance in several computer vision tasks, such as image classification [30][44], image-segmentation [33][50], object detection [18][39], and image captioning [46] [27][48][31][28][15]. CNNs-based vision systems are currently being used by the major technological companies such as, Google, IBM, Microsoft, Facebook, Twitter and Adobe. This success is attributed to the research on new activation functions such as ReLUs [19], new regularization methods like dropout [41] and the efficient use of GPUs, which allow us to train current models of 10-30 layers with million of weights 10-20x faster [9]

3.1 Convolution and cross-correlation

Since the convolutional operation represents a fundamental operation in CNNs, it's important to have an intuition behind it. Thus, we will briefly expose several properties of the convolutional operation, and the closely related cross-correlation operation. However, none of the results presented here are theoretically rigorous, and we constrain our results to only show those that are relevant for the construction of CNN architectures. Further details on convolutions can be obtained from more specialized literature on functional analysis.

Following [5], we define convolution and cross-correlation correspondingly as:

$$(K * I)(t) \equiv \int_{-\infty}^{\infty} K(\tau) I(t-\tau) d\tau \tag{28}$$

$$(K \circledast I)(t) \equiv \int_{-\infty}^{\infty} K^*(\tau) I(t+\tau) d\tau \tag{29}$$

One algebraic distinction between these two operations is that the convolution operation holds commutativity and associativity, while the cross-correlation does not [5]. This distinction does not seem to be relevant for the implementation of CNNs [4]; therefore, the CNN research community and most machine learning libraries, often call both operations a convolution, and perform either of these in their architectures. An example of this naming convention can be better understood by looking at the API documentation in theano, in which *theano.tensor.nnet.conv2d* function takes as argument the *filter_flip* flag, which results in using the same function name for both the correlation and the convolution operations.

In order to gain intuition about the convolution operation used in CNNs, we will begin by giving an example of a discrete one-dimensional cross-correlation, followed by an example of discrete one-dimensional convolution. We will observe that one can think of a convolution on real-valued functions, as a cross-correlation with a flipped kernel.

As in [24] we define the discrete 1D correlation as:

$$(K \circledast I)(i) \equiv \sum_{n=-N}^{N} K(n) I(i+n) \tag{30}$$

This definition takes into consideration that the arguments of the kernel function K begin with negative integers, while the arguments of the function I start from 0 and end at m. Any argument taken below or above the original dimensions of I would have to be defined.

In the specific case in which K is a three dimensional real-valued vector, then by the definition given, the arguments of K must be $\{-1, 0, 1\}$; this means that $K(-1)$,$K(0)$ and $K(1)$ are the components of the vector K which hold real-valued numbers. Now we consider I as a m-dimensional vector where every component could be interpreted as the value of a time series at every time step. As mentioned before, in order to be consistent with the definition, the arguments i of the vector I start at 0 and end at m. Taking into account all these considerations, we can now apply the correlation operator by running the indices $n = \{-1, 0, 1\}$ and $i = \{0, 1, ..., m\}$. This gives us the following equations:

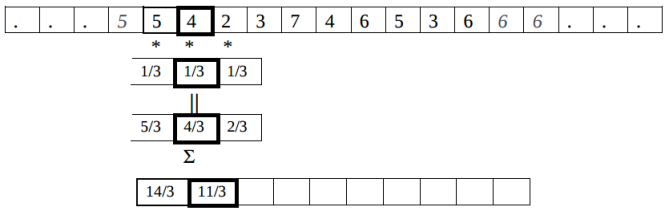


Figure 5: [24] Example of a discrete 1D cross-correlation.

$$\begin{aligned}(K \circledast I)(0) &= K(-1) * I(-1) + K(0) * I(0) + K(1) * I(1) \\ (K \circledast I)(1) &= K(-1) * I(0) + K(0) * I(1) + K(1) * I(2) \\ &\vdots \\ (K \circledast I)(m) &= K(-1) * I(m-1) + K(0) * I(m) + K(1) * I(m+1)\end{aligned} \tag{31}$$

We observe that we have not defined the value of $I(-1)$ in the first equation and the value of $I(m+1)$ in the last equation. Often in discrete convolutions and cross-correlations this values are taken as zeros, this process is called zero-padding. We can also define for any outside value of I, the value of its closest border, e.g. $I(-1) = I(0)$ and $I(m+1) = I(m)$. Note that we can also run the indices i from 1 to m-1; therefore, alleviating us from defining any values outside of I; however, by doing this we would have an output vector which would be smaller than I. When constructing large CNNs, we often leave any dimensionality reduction to a set pooling layers [4] [16], which will be later described in detail. Finally, we could also *skip* s indices in i; consequently, performing the convolution at every s step. The parameter s is called stride number, and by defining it greater than 1, we would also obtain an output smaller than I. Typical CNNs take a stride number of 1 or 2 [16]. Finally, we can observe that cross-correlation performs at every component of I, a weighted sum of the nearby elements of I, using as weights the values $K(n)$ [24]. This operation is often used for measuring similarity between functions [24].

Now we will observe how the convolution operation works for the same parameters. From [4] a discrete 1D convolution is defined as

$$(K * I)(i) \equiv \sum_{n=-N}^{N} K(n) I(i-n) \tag{32}$$

By using the same dimensions as in the cross-correlation example, and

running the same indices, we obtain the following equations:

$$\begin{aligned}
(K*I)(0) &= K(-1)*I(1)+K(0)*I(0)+K(1)*I(-1)\\
(K*I)(1) &= K(-1)*I(2)+K(0)*I(1)+K(1)*I(0)\\
&\vdots\\
(K*I)(m) &= K(-1)*I(m+1)+K(0)*I(m)+K(1)*I(m-1)
\end{aligned} \tag{33}$$

We can observe from these equations, that the multiplications between the values of K and I are crossed when compared with the ones in the cross-correlation. For example, by taking the second equation; $(K*I)(1)$, the last value of K; K(1), is multiplied by the first value of I; $I(0)$, and the first value of K; K(-1), is multiplied with the last value of I, I(2). Which is the reverse order in which it was multiplied with a cross-correlation. Therefore, if we flip the values of the kernel and perform a cross-correlation operation, it would be same as performing a convolution [24]. Subsequently, one can think of a convolution as a weighted sum of the nearby elements of I, using as weights the values of $K(-n)$, making both operations identical if the kernels are invariant to this flipping.

The 2D discrete convolution operation is homologous to the one-dimensional case, as seen in the equation below. Also figure 6 shows the explicit calculation without flipping the values of the kernel, and figure 7 shows the actual output of applying a convolution operation to an image.

$$(K*I)(i,j)=\sum_{m=-M}^{M}\sum_{n=-N}^{N}K(m,n)I(i-m,j-n) \tag{34}$$

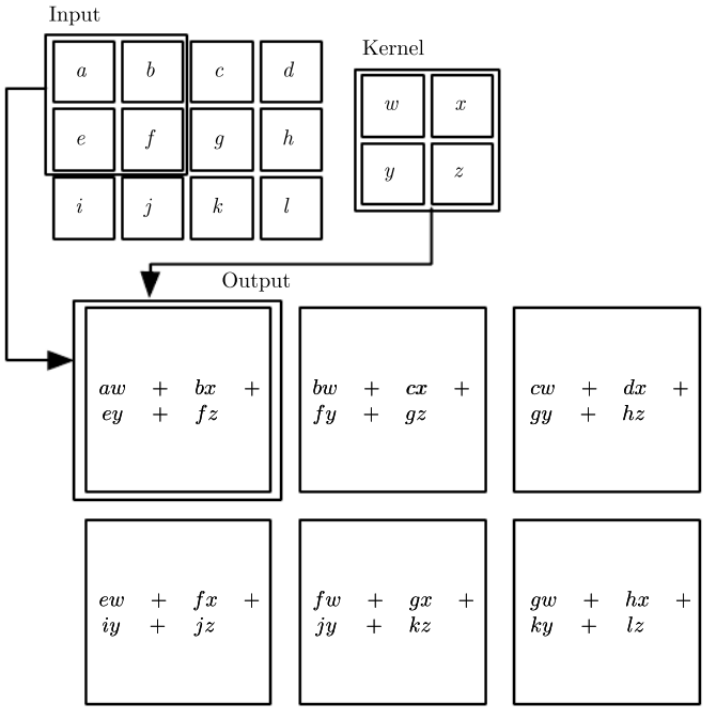


Figure 6: [4] As mentioned before, the kernel usually has a much lower dimension than the input.

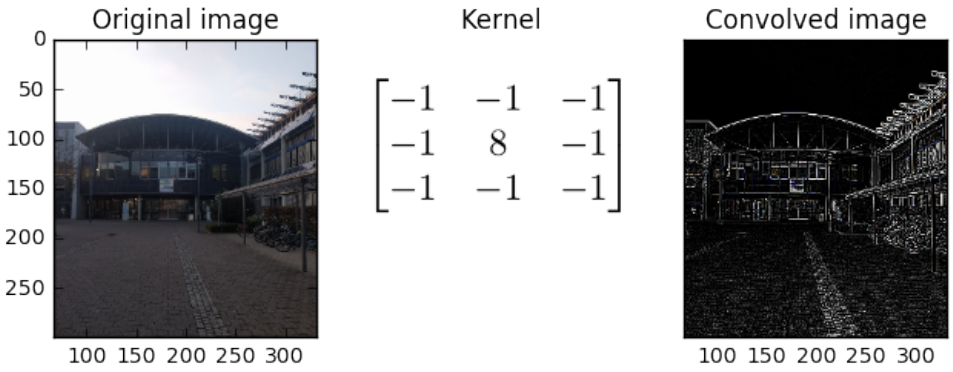


Figure 7: From left to right, the input image, the kernel parameters (weights), and the output after applying the 2d discrete convolutions.

3.2 Architecture

In this section we describe each of the elements that compose a CNN architecture. Since most CNNs process its input sequentially; like a normal MLP architecture, the composition of its elements are also considered to be layers that work in a feed-forward manner. Hence, each layer takes the value of the previous layer, modifies it accordingly to its defined operation, and outputs what is going to be the input for the next layer.

The most typical layers used in a CNN are: convolutional, activation, pooling, dropout and dense layers. We will explain all of them in detail, and conclude with general considerations that one should take into consideration while constructing CNN architectures.

3.2.1 Convolutional layer

So far we have understood how the convolution operation works on 1D and 2D discrete functions. Now we will explain how the convolution operation is incorporated in a CNN architecture meant for image processing. A CNN architecture for the purpose of image processing will be operating with a 4th-order tensor, in which the dimensions correspond to the batch number, the depth (or activation volume, or number of kernels), the width and the height. The batch number, holds a group of input examples and is used for the optimization algorithm, i.e. stochastic gradient descent (SGD). We will omit this dimension for now, and refer only to the other three.

A typical input to a CNN is an RGB image. An RGB image consists of three stacked matrices, every matrix of dimension $W_1 \times H_1$ holds discrete pixel values between 0-255, and every matrix corresponds to one of the three color channels. Therefore the input to our CNN could be represented by a 3-th order tensor of the form $[3,W_1,H_1]$. In a CNN the convolution is performed by convolving every image channel with a sliced kernel that originally has the same depth as the input tensor at that layer. In other words, for our example of the input image, we will have one kernel of dimensions $[3,F,F]$ ($F \times F$ represents the kernel size and is a hyper-parameter also called receptive field), and for each i-th dimension in the input tensor $[i,W_1,H_1]$, we will perform a convolution using the kernel $[i,F,F]$ with the input tensor. Once we have calculated the output of all these three convolutions (three 2nd-order tensors), these tensors will get summed; as defined in a typical matrix sum; therefore, resulting in an output of dimension $[W_2,H_2]$. Finally, we will sum to every element of this tensor a constant value called bias. The output after performing all this operations is called *feature map*. This whole operation has as an output a single feature map; however, we can have multiple feature maps by repeating the same procedure using different $[3,F,F]$ dimensional kernels. The number of kernels used, and consequently the number of feature maps wanted, is another hyper-parameter which we

will denote by d (depth or activation volume). By taking into account all hyper-parameters, the final output of a convolutional layer is a tensor with the following dimensions [d,W_2,H_2].

The original width and height get modified to [W_2,H_2] according to the hyper-parameters values: kernel size (F), zero-padding and stride number (S). As give in [16] the equation that relates the output dimensions with the hyper-parameters is

$$\begin{aligned} W_2 &= \frac{(W_1-F)}{S}+1 \\ H_2 &= \frac{(H_1-F)}{S}+1 \end{aligned} \tag{35}$$

As we have explained before, we want like to transform our raw input data by performing a set of transformations which work in a hierarchical order. Therefore, we can have to extend our definition of a convolutional layer, and instead of working only with the input values of dimensions [3,W_1,H_1] we need to define them for any set of dimensions [d,W_1,H_1] in which the dimension of d could hold higher dimensional input data, or a set of d feature maps outputted by a previous convolutional layer. This is done by defining kernels of dimensions [d,F,F] for our input tensor of dimensions [d,W_1,H_1], and performing the same operations. Therefore, for an input of the form [d_1,W_1,H_1] we will have a predefined number of kernels (d_2 kernels) and each of them will have a dimensions [d_1,F,F]. Each kernel will perform a convolution using its i-th dimension [i,F,F] with the i-th dimension in the input tensor [i,W_1,H_1], then all d_1 convolutions will get summed using normal matrix summation. Finally we will sum to all the elements of the 2nd-order tensor a constant value (the bias). By repeating this procedure for all d_2 kernels we will have an output tensor of dimensions [d_2,W_2,H_2]. Figure 8 explains visually our previous discussion on how convolutional layers operate in CNNs.

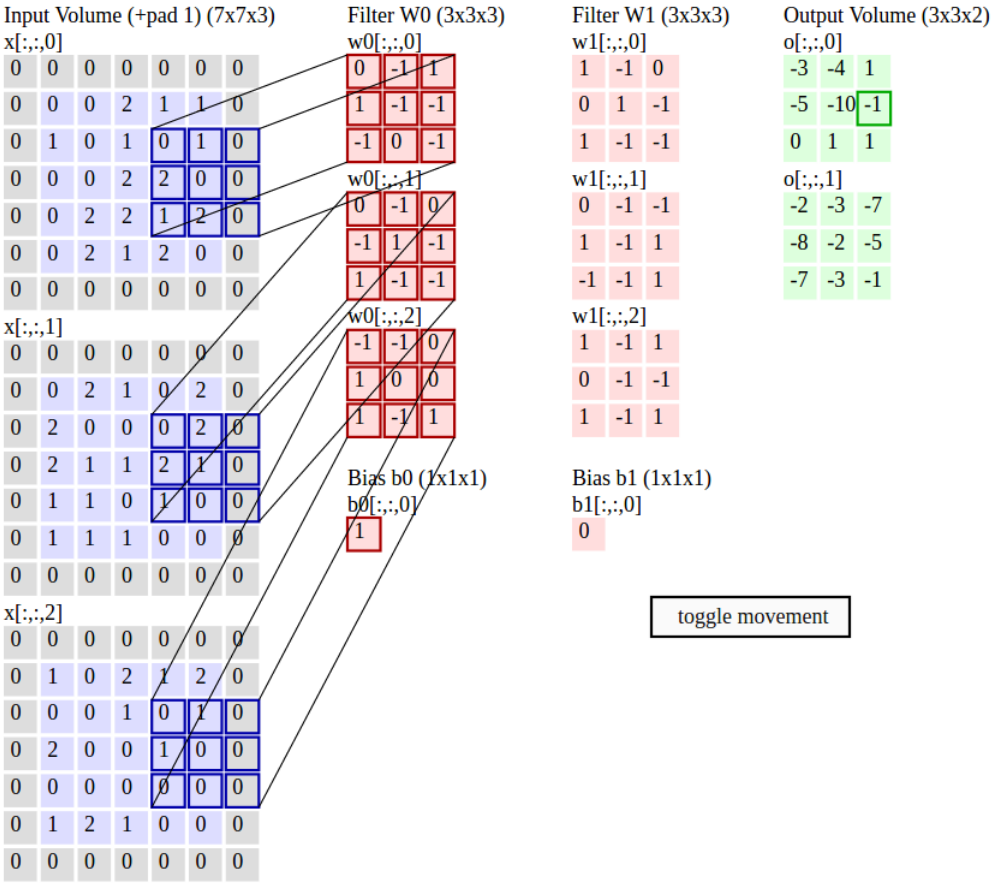


Figure 8: [16] An example of a CNN with three input dimensions, a single zero padding frame, and two feature maps, kernel size 3×3 and stride number of two.

As mentioned in the introduction, there are four ideas behind CNNs: local connectivity, parameter sharing, equivariant representation and the use of a hierarchical representation. We are now in a position to explain this properties in detail, and also extend the terminology used for artificial neural networks in the context CNNs. In order to explain these properties more we will consider a concrete example of an input image of dimensions [3,32,32], and define an architecture using a naive neural network approach. In this case we can start by multiplying each of these 3072 input values with its unique corresponding weight, then summing all multiplications (this is typically expressed as a dot product dot between an input vector of dimension 3072 and another weight vector of the same dimension), and finally applying a non-linear function to the sum; this process can be repeated for a set of

neurons in the given layer. However, one can make the assumption that image features are more correlated between small neighborhoods of pixels; a pixel in one corner may not be as highly correlated with a pixel in the middle. Therefore, instead of processing the whole input space with each neuron, we can divide it into several patches and process each patch with a separate neuron. Using our example, this mean that a neuron only takes a fraction of the 3072 input vector, e.g. a vector of spatially close values of size 3x3, and that we can have a neuron for every possible patch; taking into consideration that the patches can overlap each other. One can then naturally arrange the architecture as a 3D set of neurons, one neuron for every overlapping patch and for every dimension i-th of our input [i,32,32]. Thus, local connectivity is making the assumption that neighboring pixels are highly correlated and modifies the neural network architecture to account only for connections between neighboring values. Now that we have a set of neurons arranged in a three dimensional form; each with its own set of weights, we will make another assumption [16] : If an extracted feature is useful for some spatial position in the image, then it could also be useful at another position in the image. This means that we can try to extract the same feature by setting all the neurons weights with the same values. This is what we will refer to as parameter sharing (sparse connectivity) [4]. By considering both assumptions, we arrived to the same definition of the convolution operation applied to 3D discrete inputs, which was previously defined without any intuition for a CNN architecture.

By considering parameter sharing, local connectivity we obtain a sparse interaction, which means that for any given input of dimensions $[d_1,W_1,H_1]$ we can only have to store weights equal to the number of entries in all the kernels ($(F \times F) \times d_1 \times d_2$) (kernel size $\times$ depth of each kernel $\times$ the number of kernels).

3.2.2 Activation layer

The activation layer follows the same concept as the activation functions used in MLPs. Therefore, it applies a non-linear, often differentiable function to the sum of the weighted inputs. Since the weighted inputs are located in the feature maps, the activation can be seen as a point-wise application of this function to the feature maps. Typical activation functions used for CNNs are the ones also used for MLPs: sigmoid function, hyperbolic tangent and rectified linear units ReLUs.

3.2.3 Pooling layer

Pooling layers make a statistical summary of spatially connected neighborhoods located at the same depth [4]. One of the most common operations performed in a pooling layer is *max-pooling*. Max-pooling consists of retain-

ing only the maximum values from mutually exclusive neighborhoods in the feature maps, as seen in figure 9.

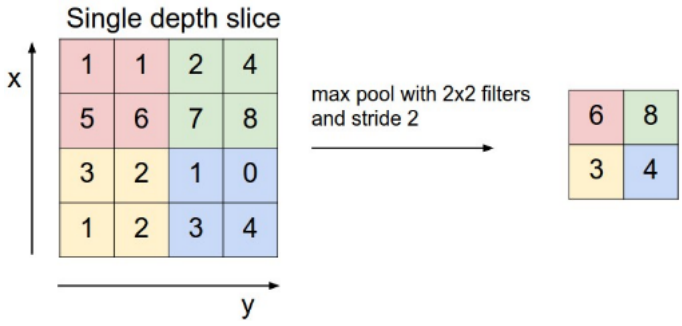


Figure 9: [16] Max pooling operation applied to a single depth slice of a feature map.

This operation on mutually exclusive neighborhoods on the same depth, is mainly used for three reasons; the first one being convenience, since it's easier to control the down-sampling of the feature maps by sampling certain values, rather than modifying the stride, the zero padding and kernel size on the convolution layer. An example of how the initial feature maps are down-sampled can be seen in figure 10. The second reason is spatial invariance. We would like to have a neural network model that is invariant to certain transformations, such as translations, rotations or even any other affine transformation. Max-pooling layers gives a small translation invariance to the network, since we are operating with a translation-invariant statistical summary on feature maps [4]. The last reason is that max-pooling only keeps this summary and discards the rest of the values; therefore, it acts as a statistical down-sampler that represents the data more concisely, and consequently more computationally efficient [4]. Other pooling functions used in practice are the mean of the neighborhood, and weighted average based on the euclidean distance from the central value of the neighborhood.

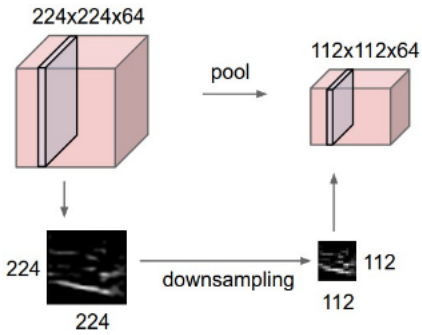


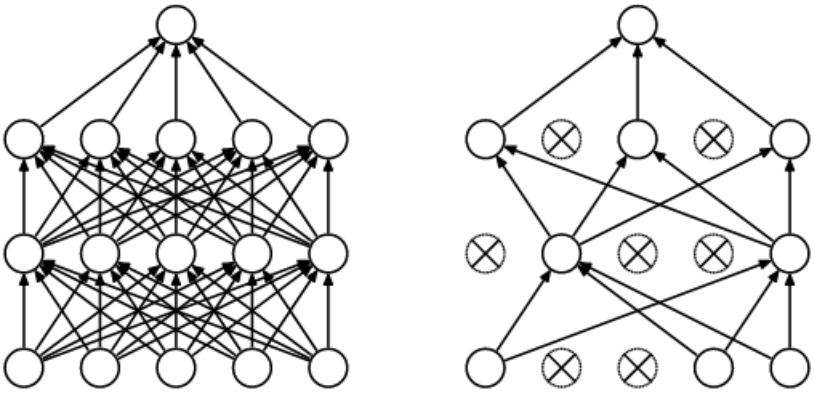
Figure 10: [16] We can observe how a pooling layer downsamples the initial input.

3.2.4 Dense layer

A dense layer is simply a normal MLP architecture, with or without a hidden layer. Therefore, it adds a layer of fully connected perceptrons. Dense layers connect with feature maps inputs by *flattening* them; this means that the whole set of feature maps will be converted into a single vector equal to $d \times W \times H$, where W and H refer to the width and height of the last feature maps, and d to the number of features maps. Dense layers are used to combine in non-linear manner the features learned by the previous layers in the convolution network. Therefore, they are used at the end of the architecture, and consists of an input layer that takes as the input the flattened features maps, followed by a hidden layer and finally and output layer. In the case of a classification problem, the output layer has the same number of neurons as categories and it uses a softmax function as the last activation. The purpose of using a softmax as activation function in the last layer is to set all values within a range of (0,1) that add all up to 1. Thus they represent the probabilities of the input value of the CNN, being in one or more categories.

3.2.5 Dropout layer

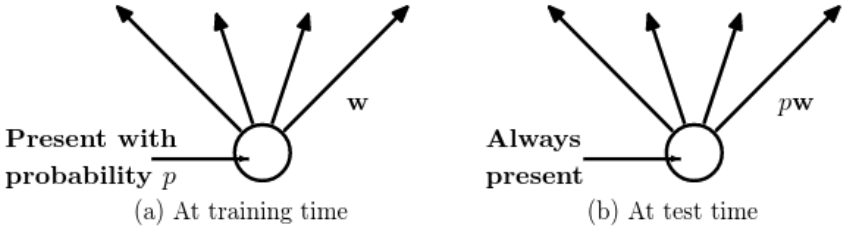
Dropout is a regularization technique that is relatively easy to implement in large-scale neural networks, and since its introduction, it has improved the performance of several neural networks models for different supervised learning tasks [41]. Therefore, it has proven to be an important technique even tough its use is complementary.



(a) Standard Neural Net (b) After applying dropout.

Figure 11: [41] **Left:** A standard MLP architecture with two hidden layers. **Right:** A thinned version of the left MLP network by applying the dropout method.

The main idea behind dropout is to randomly *turn-off* (dropout) units and their outgoing connections for every sample during its training phase, and then use the whole architecture with smaller weights in test phase [41]. This can be exemplified in the figure 11. Different weights are turned off for every sample in the training phase. It can also be tough as training *thinned* versions of the original neural network architecture, and test using the original unthinned network; however, the original unthinned weights get multiplied by the probability for which they were set to be active. This is shown in figure 12.



(a) At training time (b) At test time

Figure 12: [41] **Left:** In the training phase the neuron is set to be active; therefore, connected with neurons in the subsequent layer with a given probability p. **Right:** In the test phase the neuron is always present but the weigths connected to that neuron are multiplied by the given probability at the training phase p.

The dropout technique prevents complicated co-adaption between neu-

rons, and it was inspired by the success of sexual reproduction over asexual reproduction [41]. Sexual reproduction combines genes from two individuals and then applies a small random mutation, while asexual reproduction creates an offspring by only doing mutation in its own genes. One might be inclined to think that natural selection prioritize individual fitness; however, it seems that better fitness comes from the ability of genes to work better with another set of random genes; which eventually makes them more individually robust. Srivastava in [41] explain their motivation by stating the following: "Similarly, each hidden unit in a neural network trained with dropout must learn to work with a randomly chosen sample of other units. This should make each hidden unit more robust and drive it towards creating useful features on its own without relying on other hidden units to correct its mistakes. However, the hidden units within a layer will still learn to do different things from each other"

3.2.6 Architecture composition

So far we have mentioned the operational details of several layers, but we have not explained how to combine them to create a successful CNN model. Here, we will only refer to general CNN architectures applied to a classification task. The typical composition of layers include several convolutions layers; between 1 to 3 layers, followed by an activation layer, then a pooling layer, and optionally a drop-out layer. This composition of layers is repeated several times depending on the application. We then proceed by flattening the feature maps and use 1 to 2 dense layers. The final dense layer should have the same amount of neurons as categories in the classification, and it should use softmax as its activation function. Typical kernel sizes are $[3 \times 3]$, $[5 \times 5]$ or $[7 \times 7]$ pixels. Most of the current architectures use ReLus as their activation function. Usually, the max-pooling layers operate on pixel neighborhoods of sizes $[2 \times 2]$ or $[4 \times 4]$. Dropout layers can be included after activation layers, and as mentioned in [41] a good starting value for their activation probability is .5. Currently most of these hyper-parameters are obtained empirically through cross-validation.

3.3 Back-propagation for CNNs

The final result of the back-propagation algorithm applied to MLPs showed that we can optimize the weights in layer l by propagating the error from the last layer L backwards into previous layers until we arrive to layer l. Since we can transform any MLP into CNN by taking into consideration assumptions that only modify the number of weights and their connections, then the back-propagation algorithm should be calculated similarly as for a MLP architecture. This is visualized in figure 13.

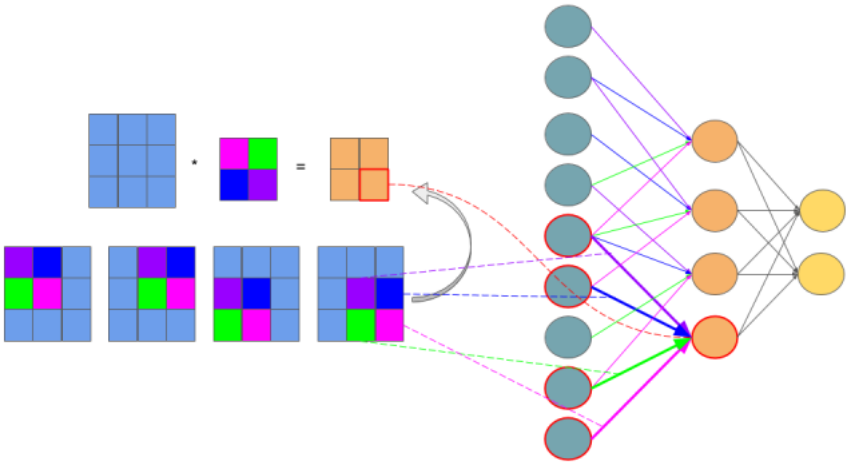


Figure 13: [23] Following the assumption of local connectivity and parameter sharing, we created a specialized architecture for grid-like topologies such as images. A visualization of this assumptions being realized, and consequently transforming a MLP into a CNN can be here observed. First we can observe that the kernel weights are flipped in both spatial directions as defined in the convolution operation. The blue rectangles in the left side of the image are represented in a flattened manner in the right side of the image. This flattening version exhibits the two constrains: first, not all neurons in the first layer are connected with the next layer. In fact the neurons in the orange hidden layer always receive four inputs which correspond the local patches in the blue rectangle. The second property is that these local weights are shared; in other words, we do not use different weights for every local patch. We can visualize this result by identifying that the weights; which are represented with colors, are repeated.

Using the same image above we can now visualize the back-propagation algorithm by assigning an error to the hidden layers. As defined in previous sections, the backward pass will weight this errors by multiplying them with the weight that connects them. Figure 14 hints into a more elegant arrangement of the back-propagated errors. We will later demonstrate that the error of each neuron can be easily calculated by convolving the errors from top layers with spatially flipped kernels.

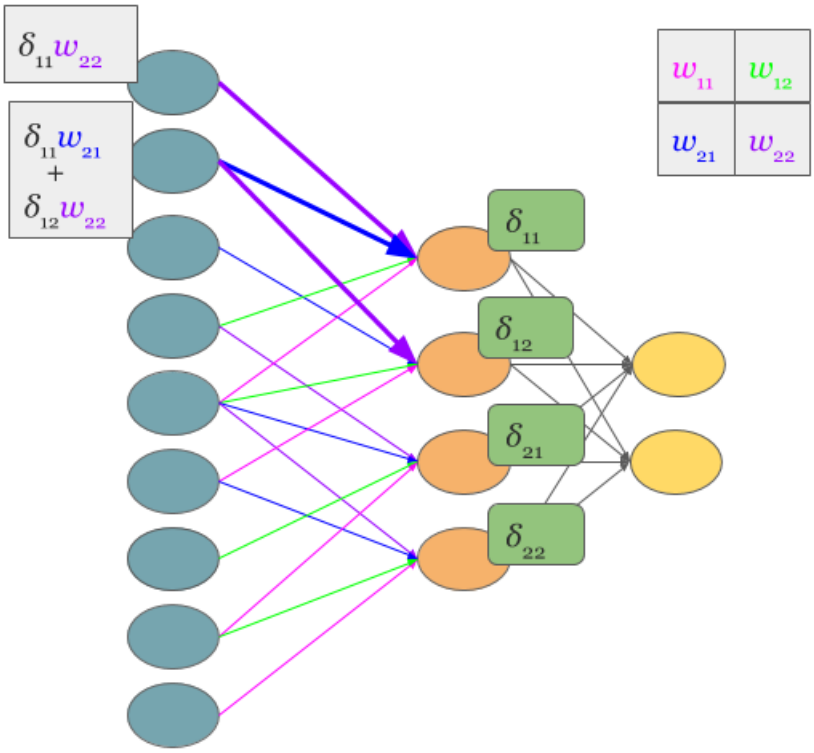


Figure 14: [23] As in MLPs the error associated to the first blue layer only depend on errors and weights that connected them in the forward pass with higher layers. However, by imposing local connectivity and weight sharing the amount of connections and therefore errors dropped considerably.

Following [23] we will now formalize the back-propagation algorithm for CNN architectures with a single feature map; however, this can be easily generalized by summing the gradients over all feature maps.

We defined in the ANN section the output of each neuron j in layer l as z_j^l, and it takes the following known form

$$z_j^l = \sigma_j^l(net_j^l) \tag{36}$$

The local connectivity and parameter sharing assumptions transform the dot product in net into a convolution. Also, since we are dealing with a set of neurons that got converted into a 2D structure, we change the indices from net_j to $net_{x,y}$. Consequently, we end up having the following equation:

$$net^l_{x,y} = (w^l * x^{l-1})(x,y) + b^l_{x,y} = \sum_{m=-M}^{M} \sum_{n=-N}^{N} w^l_{m,n} x^{l-1}_{x-m,y-n} + b^l_{x,y} \quad (37)$$

One important aspect that should be mentioned is that we can no longer "hide" the bias term in the weight's first parameter, and now we show it explicitly.

As previously did in the ANN back-propagation, we now substitute the input values coming from the previous layer $x^{l-1}_{x-m,y-n}$ with the explicit value calculated at the previous layer

$$net^l_{x',y'} = (w^l * \sigma^{l-1}(net^{l-1}))(x',y') + b^l_{x',y'} = \sum_{m=-M}^{M} \sum_{n=-N}^{N} w^l_{m,n} \sigma^{l-1}(net^{l-1}_{x-m,y-n}) + b^l_{x',y'} \quad (38)$$

CNNs follow the same composition of functions as a typical MLP; therefore, the partial derivatives follow the same order, but obviously its calculation differ since we are introducing the convolution operator. This lets us calculate the gradient of the cost function as we did previously for the MLP architecture, and consequently define on the same manner the neuron's error for intermediate layers. We refer only to the intermediate errors in the layers, since most of the common CNN architectures locate the convolutional layer in the middle, as explained in the previous section.
Therefore, we define the neuron's error at intermediate layers in a CNN with a single feature map as:

$$\delta^{l-1}_{x,y} = \sum_{x'} \sum_{y'} \delta^l_{x',y'} \frac{\partial net^l_{x',y'}}{\partial net^{l-1}_{x,y}} \quad (39)$$

Substituting the value of $net^l_{x',y'}$ for its actual convolution we obtain the following

$$\delta^{l-1}_{x,y} = \sum_{x'} \sum_{y'} \delta^l_{x',y'} \frac{\partial (\sum_{m=-M}^{M} \sum_{n=-N}^{N} w^l_{m,n} \sigma^{l-1}(net^{l-1}_{x'-m,y'-n}) + b^l_{x',y'})}{\partial net^{l-1}_{x,y}} \quad (40)$$

As explained in the back-propagation section of MLPs, we notice that all partial derivatives are equal to zero except when $x = x' - m$ and $y = y' - n$.

$$\delta^{l-1}_{x,y} = \sum_{x'} \sum_{y'} \delta^l_{x',y'} w^l_{m,n} \sigma'^{l-1}(net^{l-1}_{x,y}) \quad (41)$$

Looking closely at the equation above, there are restrictions regarding the indices m, n, x, y, x', y'. Those restrictions are the ones given by the values for which the derivatives are not zero $x = x' - m$ and $y = y - n$. We can incorporate those restrictions by substituting two indices since two restrictions were incorporated, and we do so by taking $m = x' - x$ and $n = y' - y$. This substitution leads to the following result

$$\delta^{l-1}_{x,y} = \sum_{x'} \sum_{y'} (\delta^{l}_{x',y'} w^{l}_{x'-x,y'-y}) \sigma'^{l-1}(net^{l-1}_{x,y}) \tag{42}$$

This last expression can also be interpreted as a convolution of spatially flipped kernels. This results comes from the fact that we can change $w^{l}_{x'-x,y'-y}$ into $w^{l}_{-(x-x'),-(y-y')}$ in order to satisfy the convolution operator definition, this lead us to the following equation

$$\delta^{l-1}_{x,y} = (\delta^{l}_{x,y} * w_{-x,-y}) \sigma'^{l-1}(net^{l-1}_{x,y}) \tag{43}$$

With this equation we can calculate the errors made between neurons in convolutional layers. However, in order to calculate the complete gradient of the cost function with respect to these set of weights we are missing the calculation of the last partial derivative, since the neuron's errors were defined from the first partial derivative to the penultimate one. The form of this last partial derivative changes since we also changed the definition of net.

From previous sections we know that the last partial derivative is taken with respect to the weights we want to calculate the update for. For CNN we have to calculate the following

$$\frac{\partial net^{l-1}_{x,y}}{\partial w^{l}_{m,n}} = \frac{\partial (\sum_{m'} \sum_{n'} w^{l}_{m',n'} \sigma^{l-1}(net^{l-1}_{x-m',y-n'}) + b^{l}_{x,y})}{\partial w^{l}_{m,n}} \tag{44}$$

We again observe that all the derivatives are zero except when $m = m'$ and $n = n'$

$$\frac{\partial net^{l-1}_{x,y}}{\partial w^{l}_{m,n}} = \sigma^{l-1}(net^{l-1}_{x-m,y-n}) \tag{45}$$

By putting all partial derivatives together; and therefore completing the gradient of the cost function, we can obtain the update value for each neuron

$$\Delta w^{l}_{x,y} = \delta^{l-1}_{x,y} \sigma^{l-1}(net^{l-1}_{x-m,y-n}) \tag{46}$$

This result can also be seen as a convolution using flipped values for net. Finally we arrive to the update equation for the back-propagation algorithm using CNN architecture

$$\Delta w^{l}_{x,y} = \delta^{l-1}_{x,y} * \sigma^{l-1}(net^{l-1}_{-x,-y}) \tag{47}$$

Now we can just substitute the equations for the neuron's error and the weight update calculate here for a CNN, instead of the ones used in the MLP, and use exactly same back-propagation algorithm presented in ANN section,in order to minimize the cost function.

4 LSTM architecture

The long short-term memory (LSTM) architecture is a recurrent neural network that incorporates in every neuron operations that mimic memory chips [21]. Such operations are write, read, and reset, and are represented in the LSTM as differentiable gates that control the output, input, and the feedback of previous sates in the neuron. In other words, one can think of an LSTM as a differentiable memory chip used in digital computers [21].

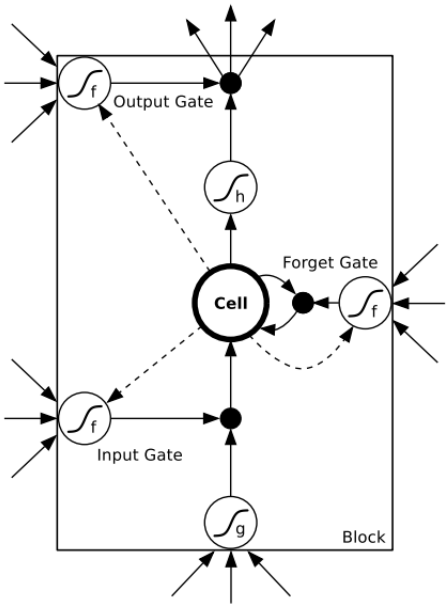


Figure 15: [21] Single LSTM neuron with a single cell. The peephole connections are represented by the dashed lines.

We will now proceed by reviewing the operations inside a single neuron, such as the one visualized in figure 15. For now we will omit mentioning all incoming connections to the input, forget and output gates. These connections will be later described when we review a whole LSTM architecture with more neurons.

As in typical ANNs the input values are connected to the lower part of the neuron. These inputs are multiplied by their corresponding weights; then,

the weighted inputs get summed and an activation function g is applied to them. This input value is then controlled by the input gate, which dictates how much influence this input value will have in the neuron. The input gate controls the input by multiplying it with a value between zero and one. The value of the input gate is obtained by summing up all its incoming weighted connections, and then applying a sigmoid function to them. This controlled input is then passed to the cells inside the neuron. In the figure above we are only depicting a single cell; however, other architectures may include more cells inside each neuron. The cell inside the neuron is connected to the output of the input gate and with a recurrent connection that is controlled by the forget gate. The forget gate also takes its incoming inputs, multiplies them with a set of weights and finally applies a sigmoid function. Consequently, the forget gate controls how much influence will previous cell states have on this result. The sum of the controlled input and the controlled recurrent values of previous states, is what call the current cell state. We then apply a function h to the current cell state. Finally the output of the neuron is controlled by the output gate. Naturally, this output gate is again formed by applying a sigmoid function to the weighted sum of its inputs. One can think of the output gate as controlling the amount of information that gets written.

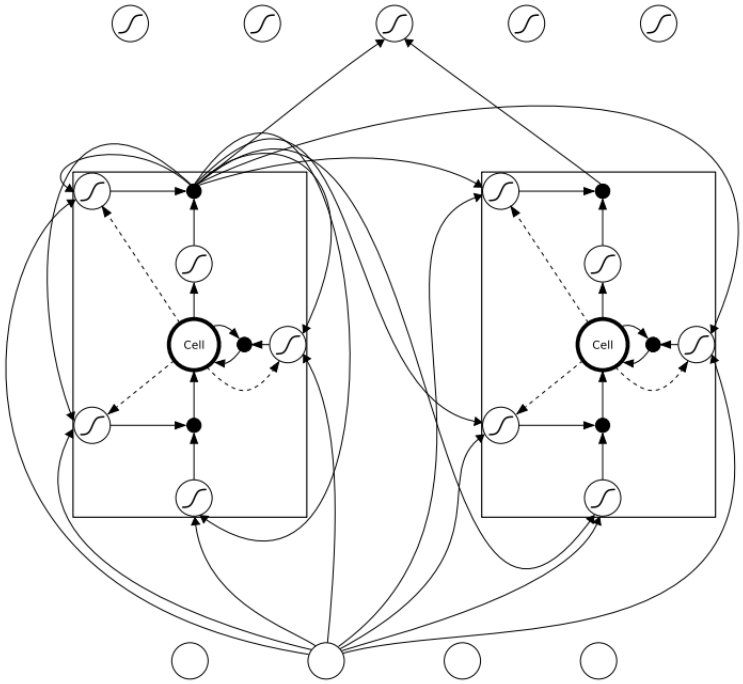


Figure 16: [21] LSTM architecture. Not all the connections are displayed.

We will now discuss the structure of a complete LSTM network as displayed in figure number 16. Also, we will define all the input connections to the neuron and to its gates. As seen in the LSTM network in figure 16, each LSTM neuron has four inputs but a single output. These four inputs are: the neuron's normal input, the input gate, the forget gate and output gate. The single output of each neuron is the only result that gets propagated to the rest of the network [21]. Before we proceed any further we will clarify some of the terminology that we will be using to describe the forward and backwards pass. This terminology will be mostly based on [21]. As in previous chapters, w_{ji} will be the connections from units i to units j. We will use the subscripts ι, ϕ and ω to refer to the input gate, forget gate and output gate respectively. The subscript c will be used to refer to the cells inside the LSTM. With this terminology, $w_{\iota c}$, $w_{\phi c}$ and $w_{\omega c}$ refer to the weights that connect each cell in the LSTM with the input gate, output get and forget gate respectively. This connections are called *peephole weights* since they allow the gates to look inside the neurons state [17]. The state of each cell c at time t will be referred as s_c^t. As mentioned before, the state of each cell is just the sum of the controlled input and the controlled previous state. Finally, we will define I, K, H and C respectively as the number of input values, output values, hidden neurons, and cells inside each neuron.

We will describe now the forward pass of single LSTM neuron. The input gate receives the following inputs:

$$net_\iota^t = \sum_{i=1}^{I} w_{\iota i} x_i^t + \sum_{h=1}^{H} w_{\iota h} z_h^{t-1} + \sum_{c=1}^{C} w_{\iota c} s_c^{t-1} \tag{48}$$

The summations on the right side of the equation correspond from left to right to the input values, the output values from other LSTM neurons in the hidden layer, and the peephole connections. The connection from both the hidden neurons and peepholes are calculated from the previous temporal state $t-1$. As mentioned before, the activation function used after summing up the inputs for the input gate is a sigmoid function; consequently, we describe the actual value of the input gate as:

$$z_\iota^t = \sigma(net_\iota^t) \tag{49}$$

The forget gate follows a similar calculation as in the input gate. It takes as input the weighted values from the inputs, all hidden neurons and the peephole connections.

$$net_\phi^t = \sum_{i=1}^{I} w_{\phi i} x_i^t + \sum_{h=1}^{H} w_{\phi h} z_h^{t-1} + \sum_{c=1}^{C} w_{\phi c} s_c^{t-1} \tag{50}$$

Again, the sigmoid function is applied to this sum

$$z_{\phi}^{t} = \sigma(net_{\phi}^{t}) \tag{51}$$

As mentioned before the state of the cell equals to the sum of the final outputs of the forgot gate and the input gate. Fist we calculate the actual input to the LSTM, which corresponds to the weighted sum of the input values x_i and the output of all previous LSTM neurons in the hidden layer z_h^{t-1}.

$$net_{c}^{t} = \sum_{i=1}^{I} w_{ci} x_{i}^{t} + \sum_{h=1}^{H} w_{ch} z_{h}^{t-1} \tag{52}$$

We then multiply it by the value of the already calculated input gate that passes through a non-linear activation g, and we sum it with the controlled previous temporal state of the cells s_c^{t-1}.

$$s_{c}^{t} = z_{\phi}^{t} s_{c}^{t-1} + z_{\iota}^{t} g(net_{c}^{t}) \tag{53}$$

Similarly to the input gate we calculate the values of the output gate as:

$$net_{\omega}^{t} = \sum_{i=1}^{I} w_{\omega i} x_{i}^{t} + \sum_{h=1}^{H} w_{\omega h} z_{h}^{t-1} + \sum_{c=1}^{C} w_{\omega c} s_{c}^{t-1} \tag{54}$$

$$z_{\omega}^{t} = \sigma(net_{\omega}^{t}) \tag{55}$$

Finally, the output of each neuron with respecto to the cells in the LSTM layer are:

$$z_{c}^{t} = z_{\omega}^{t} h(s_{c}^{t}) \tag{56}$$

Where the function h is usually a non-linear activation function such as an hyperbolic tangent.

4.1 LSTM back-propagation

Back-propagation in a RNNs is usually done through the algorithm back-propagation through time (BPTT). The main idea of BPTT is to unfold the recurrent graph and interpret it in a sequential manner [21]. A unfolded graph is depicted in figure 17. Consequently, the unfolded graph will not contain any recurrent cycles. This let us apply the forward pass as defined previously, and also calculate the backward pass by using normal back-propagation.

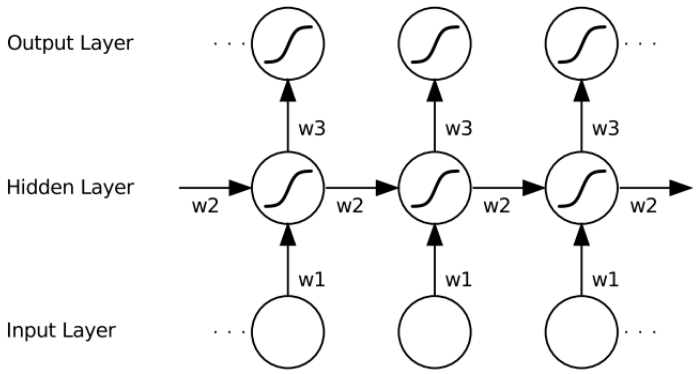


Figure 17: [21] Unfolded recurrent graph

5 Image captioning

In this section we will review the state-of-the-art approaches for image captioning, and evaluate their performance according to the standard metrics used for this natural language processing (NLP) tasks. First, we will proceed by conceptualizing the general problem, and then by dwelling into the specific details of all different approaches here reviewed.

5.1 Introduction

Describing the content of an image through grammatically correct sentences represents one of the most challenging problems in artificial intelligence. This problem merges two of the most fundamental areas in AI; computer vision, and natural language processing, and it has raised interest in the research community since the released of two large datasets, MS-COCO [32] and Flickr30k [38].

The main idea behind image captioning consists of encoding an image using a pre-trained CNN, and then decoding this extracted features into word representations using a RNN. Hence, the actual input to the network is an image, and its output is a sequence of one-hot representations of words in a vocabulary. We will start by reviewing the most typical convolutional neural network encoders used in image captioning. Then we will proceed by explaining the state-of-the-art model used for image captioning, as well as other important variations.

5.2 Convolutional feature maps

5.2.1 VGG networks

The VGG networks secured first and second places in the localisation and classification tasks respectively, for the ImageNet-2014 competition [40]. The ImageNet-2014 classification task consisted of classifying 1.2 million images over 1000 categories, for which an ensemble of 3 VGG networks obtained a 7.32% Top-5 error [12]. Also, the VGG networks were among the first CNNs that successfully implemented bigger depths; with 16 and 19 layers corresponding to the networks VGG16 and VGG19. The input of these networks is a fixed-size 224×224 RGB images, and the only pre-preprocessing done to the images is a subtraction of the mean RGB value computed over the training set. They provided a roadmap for constructing successful CNN architecture, by reducing the kernel size; consequently reducing the number of weights, incorporating dropout as a regularization technique, and using ReLUs as activation functions. The training was carried using a multinomial logistic loss function with mini-batch gradient descent with momentum. The batch size was of 256 samples and they used a momentum of 0.9. The training used a L2 regularization penalty of 5×10^{-4}, and the dropout ratio was set to 0.5. A concrete clarification of all its layers is depicted on figure 18.

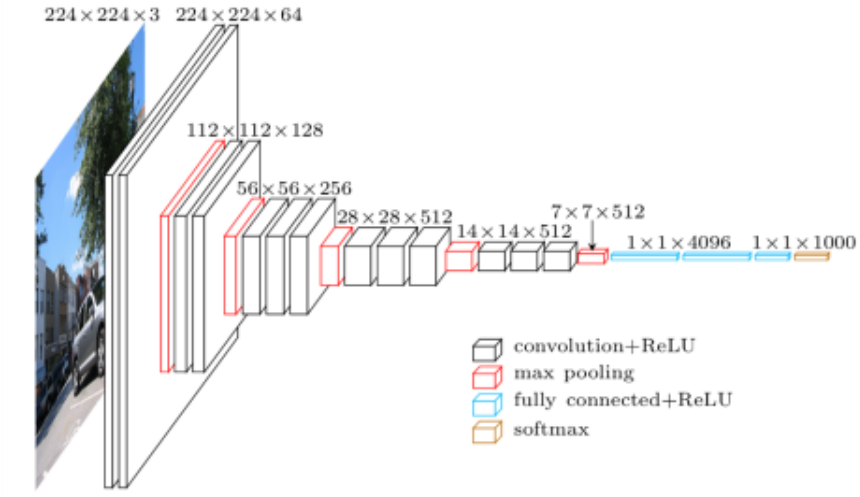
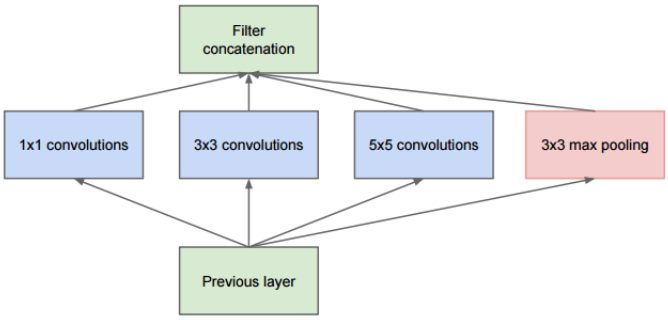


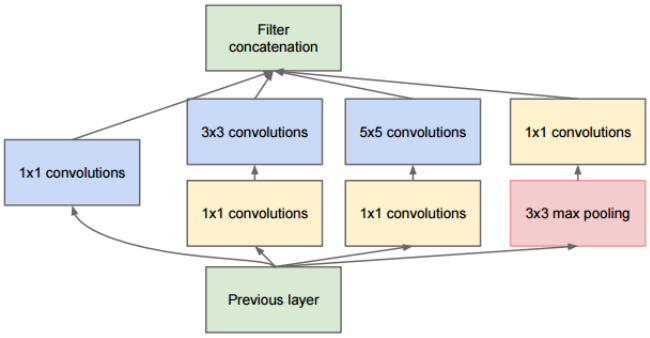
Figure 18: [10] VGG16 CNN. Note that the 16 layers do not take into account max-pooling layers; however, they do count the last three fully connected layers. The main difference between VGG16 and VGG19 is that VGG19 adds one more convolution layer to each of the last three convolution layers of VGG16.

5.2.2 GoogLeNet

Another commonly used CNN image encoding is GoogLeNet. GoogLeNet won the 2014 ImageNet classification task with a 6.65% Top-5 error [44]. Similarly as the VGG networks, GoogLeNet takes as input a RGB image with zero mean. This network uses 12 times less parameters than the winning architecture of Krizhevksy from 2012 (AlexNet [30]) while being considerably more accurate [44]. The design of the architecture is partly inspired from the practical intuition that visual information is naturally processed with different scales and then merged; hence, the next stage in the network would be able to process features with different scales simultaneously and consequently in a more efficient manner [44]. This novel construction receives the name of *Inception-module* and it can be visualized in figure 19. The Inception-module uses 1×1 convolution filters before computing the expensive 5×5 and 3×3 convolutions. This idea is based on dimensionality reduction techniques, which state that low dimensional embeddings can contain relevant information of its high-dimensional counterparts [44]. Therefore, they reduced the previous number of feature maps with the one-dimensional convolutions before applying the convolutions with bigger kernels.



(a) Inception module, naïve version



(b) Inception module with dimensionality reduction

Figure 19: [44] Inception modules. One important aspect about the inception modules is that in order to concatenate all the feature maps outputted from the convolutions and the max pooling operations, the convolutions were performed with a stride number that made the width and the height of the feature maps match the downsampled feature maps from the max-pooling operation.

Their final network consisted of 27 layers including pooling layers. As mentioned by the authors of [44], given the large depth of the network, propagating the gradients back to the initial layers was a concern. Considering that shallower networks also produce strong performances on a classification task, they added auxiliary classifiers on intermediate layers. This helped to combat the vanishing gradient problem, and to provide a regularization to the network [44]. These classifiers were smaller convolutions, that were discarded during the exploitation phase of the network.

GoogLeNet was trained with stochastic gradient descent with 0.9 momentum, and by decreasing the learning rate by 4% every 8 epochs [44]. The authors also mentioned that GoogLeNet could be trained using few high-end GPUs withing a week. Their final submission to ImageNet-2014 was an ensemble of 6 GoogLeNets.

5.3 Datasets

One of the most used datasets for image captioning is COCO (common objects in context) [32]. This dataset was introduced with the purpose of addressing new challenges: non-iconic image descriptions and classification as well as exact 2D localization [32]. The difference between iconic and non-iconic images is displayed in figure 20. COCO was specifically designed for image segmentation and captioning. An important remark of this dataset is that it provides individual segmentation between classes as observed in figure 21.

Figure 20: [32] Current computer vision tasks have performed well on iconic images; however, real-world situations are rarely iconic and do not present a single clear object.

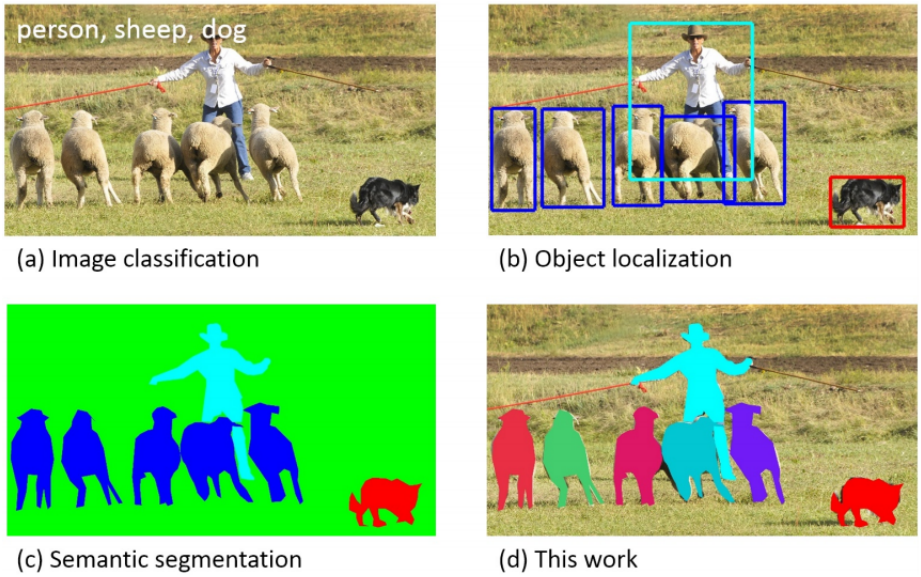


Figure 21: [32] Images (a),(b) and (c) represent typical challenges in datasets such as ImageNet [12] (d) The COCO dataset extends the difficulty by providing individual segmentations even withing the same class.

The COCO dataset contains 91 object categories, 2.5 million labeled instances and 328,000 images. One important aspect about this dataset is that it contains more object instances in every image; ImageNet contains 3 objects per image while COCO contains 7.7. COCO also contains a training set of 413,915 captions for 82,783 images, a validation set of 202,520 captions with 40,504 images and a test set of 379,249 captions for 40,775 images [6]. Almost all of the images contain at least 5 captions and they were collected and captioned using Flickr images and Amazon's mechanical turk.

The COCO dataset used 70,000 worker hours, which were predominately distributed on Amazon's mechanical turk service [32].

5.4 Metrics

The standard metrics used in image captioning are BLEU [37] and METEOR [2]. However, there has been criticism regarding BLEU since it has shown that it correlates poorly with human judgment in comparison to METEOR [2] [45] [48] [27]. METEOR calculates its score by using 3 matching components which are processed in the following order: exact, stem and synonym [2]. These components make an injective mapping between both strings. The exact component maps unigrams that are identical, the stem component maps unigram that have the same word-stem; also called root form, and the synonym component maps two unigrams if they are synonyms.

Since mappings are not unique METEOR always tries to maximize the number of matches. In the case in which two mappings have the same number of matches, METEOR selects the one that has the least amount of crosses. Crosses can be intuitively understood by aligning reference sentence and the machine generated sentence one above the other one, and drawing straight lines between the matched words. The number of crosses will then be the number of intersections between the matching connections [2]. After running the matching modules we calculate the unigram precision (P): ratio between the number of unigrams matched in the machine generated sentence and the total number of unigrams in the machine generated sentence. Also, the unigram recall (R) is calculated: ratio between the number of unigrams matched in the machine generated sentence and the total number of unigrams in the reference sentence. Then, we calculated a weighted F-score that puts more weight on the recall. In [2] they propose using the harmonic mean of $9R$ and P.

$$F_{\text{score}} = \frac{10PR}{R + 9P} \tag{57}$$

METEOR also calculates a penalty value:

$$\text{Penalty} = 0.5 * \Big(\frac{\text{chunks}}{\text{unigrams matched}} \Big)^3 \tag{58}$$

Where chunks refers to the number of adjacent positions in the machine generated sentence that are mapped also to adjacent positions in the reference sentence. In the extreme case where the computer generated sentence is the same as the reference frame we will have only one chunk. The penalty value increases when the number of chunks increases up to a maximum of 0.5. The penalty lower bound is dictated by the number of unigram matches [2]. The METEOR score is then calculated as follows:

$$\text{Score} = \text{F-score} * (1 - \text{Penalty}) \tag{59}$$

5.5 State-of-the-art model for image captioning

The current state-of-the-art model for image captioning was originally proposed in [46] and later modified by the same authors on [47]. One of its main advantages over previous models is that this architecture is a single end-to-end trainable architecture; consequently, it alleviates slow performances and complicated pipelines.

The basic idea of this approach is that given an image I, they train a probabilistic language model p; with parameters θ, so that it maximizes the likelihood $p(S|I;\theta)$, where S is a sequence of words $S = \{S_1, S_2, ...\}$. This is expressed in mathematical terms as:

$$\theta^* = \underset{\theta}{\text{argmax}} \sum_{I,S} logp(S|I;\theta) \tag{60}$$

As mentioned by authors in [46], this model was inspired on the current state-of-the-art approaches for machine translation. Those models train a probabilistic model that maximizes the likelihood of producing a sequence of words T, written in a target language, given a sequence of words S in a source language. However, maximizing the likelihood for finding the parameters of a model that best fit the data is not they key concept behind the recent advances on machine translation. They main idea was to propose a RNN as a probabilistic model, and use the weights as the parameters for which we would like to maximize the likelihood [43]. The RNN model uses a LSTM network to encode the sentence S and uses another LSTM network to decode it into a sentence T. Extrapolating these ideas for image captioning, they propose a CNN to encode the image and a LSTM to decode it into a sentence. They named this joint model *neural image caption* (NIC).

We know that given a log joint probability distribution $logp(S|I;\theta)$ we can apply the chain rule of joint probabilities over the words $S = \{S_0, ..., S_N\}$, assuming that this sentence contains N words. This results on following equation

$$logp(S|I;\theta) = \sum_{t=0}^{N} logp(S_t|I, S_0, ..., S_{t-1};\theta) \tag{61}$$

In the equation above the product between the probabilities turns into a sum since we are taking the logarithm of the likelihood. Therefore, our optimization task gets reduced to maximize the sum of these log probabilities given in the equation above [46].

A LSTM network fits naturally to this approach since we can encode information from a variable number of previous given words into the hidden state of the network [46]. The LSTM neuron proposed for this model is similar to the one presented in the LSTM section; however, it only contains a single cell C and they do not account for any peephole weights between the gates and the cell.

Now that we have obtained an overview of the general model proposed, we can review all the technical details. The image encoding was done using GoogLeNet without its last classification layer. This penultimate layer outputs a vector of 2048 dimensions which corresponds to relevant features of the image. Each word is represented using a vector of dimensions equal to the vocabulary size using a one-hot-encoding. Thus, every word is a vector which has zero in all its components except for one for which its value equals one. Now that we have a numerical representation of words and images, the authors proceeded by embedding words and images into a vector space of the same dimensions. They do this by adding a fully connected layer that takes as input the encoded image and maps it into a smaller dimension D. Similarly, they construct another fully connected layer that transforms every one hot encoding into a vector of the same dimension D. Finally, a LSTM

networks is initialized with the embedded image features and a special word-token that indicates the beginning of the sentence. This LSTM network calculates at every time step using a softmax activation function, a vector which dimension equals the number of words in the vocabulary. Therefore, this last vector assigns probability to each word at every time-step of the LSTM. The equations that describe this transformation are written below.

$$x_{-1} = W_I \text{CNN}(I) \tag{62}$$

$$x_t = W_e S_t, t \in \{0, ..., N-1\} \tag{63}$$

$$p_{t+1} = \text{LSTM}(x_t), t \in \{0, ..., N-1\} \tag{64}$$

The matrices W_I and W_S represent the fully connected layers with a linear activation function. In this setup the image I is only taken as input once. The authors had empirically tested that giving the image as input at every time step yields worse results. The LSTM network was trained to predict the next word in a sentence, given the image and the previous hidden state. Consequently the model is better expressed by unrolling the LSTM network; thus, making a copy for every word in the sentence. This visualization is displayed in figure 22.

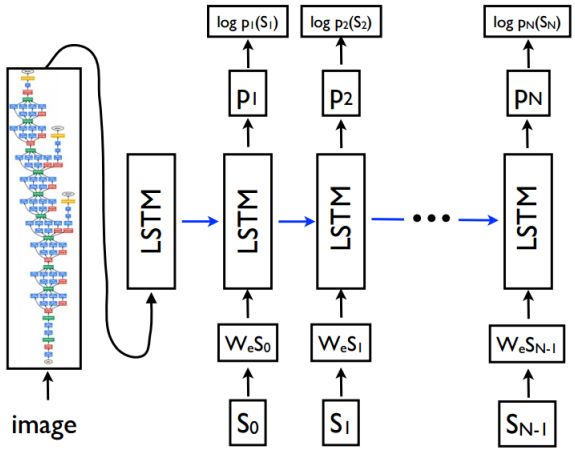


Figure 22: [46] Image captioning model used in *show and tell*. The matrix W_I is implicitly given inside the CNN.

An instantiated example of figure 22 is displayed on figure 23. Figure 23 shows an important characteristics of the LSTM model, which is that the sentences are padded with special word-tokens that indicate the beginning (BOS) and end of the sentence (EOS).

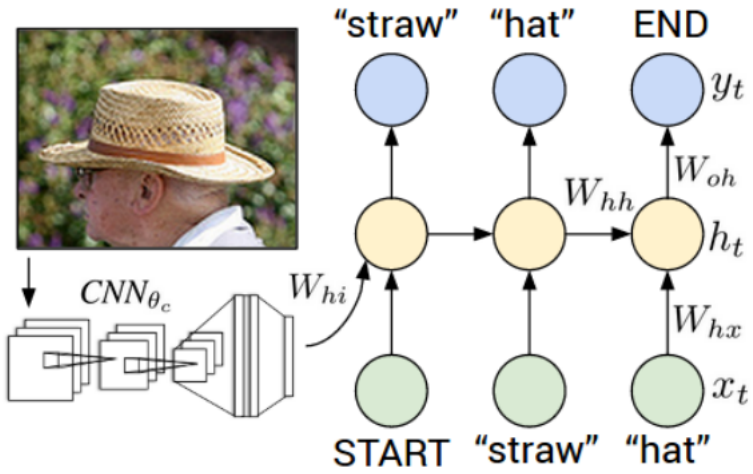


Figure 23: [28] This image shows us explicitly how could we sample from our trained model. In this specific example, we initialize our LSTM with the image features from the CNN, we then give as first word the BOS token; in this case START, we then run the model a single step and obtain the most probable word, we take this word and use it as input for the next step. We repeat this process until the model outputs the special word-token EOS (END in this case).

The authors also mention that the loss function is given directly by the equation 61 as:

$$L(I,S) = -\sum_{t=1}^{N} logp_t(S_t) \tag{65}$$

Where p_t is probability of the correct word. However, when investigating the source code directly they use the categorical cross-entropy loss which is defined in [35] as:

$$L(I,S) = -\sum_{t=1}^{N} p(S_t)log((q(S_t))) \tag{66}$$

This loss is minimized with respect to all the weights in the LSTM, and the matrices W_I and W_S. One final remark of the model is that they used a beam-search with beam size of 20. Therefore, instead of sampling the most probable word as explained in figure 23, they considered the best 20 sentences at every time-step, and at the end they only display the best constructed sentence. They trained the model using a stochastic gradient descent, with no momentum and with a fixed learning rate [46]. The dimensions used for the word embeddings were of 512 dimensions. Also, the

number of hidden neurons in the LSTM architecture were 512. Finally, the authors mention that they used basic tokenisation, and that they removed all words that had frequency less than five in the training set. Several results from their model are displayed in figure 24.

Figure 24: [46] Results grouped by human ratings.

Another important feature that can be studied from the architecture is the word embeddings learned by the model. Since the image features correlate between objects, the authors hypothesize that in the extreme case when a word is rarely used (e.g. unicorn) the model can interpret this object similarly as to its closest embedding classes (e.g. horse). This provides more information that would have been lost by a traditional method such as a bag-of-word approach [46]. Examples of several word embeddings from [46] are shown in fig 25.

Word	Neighbors
car	van, cab, suv, vehicule, jeep
boy	toddler, gentleman, daughter, son
street	road, streets, highway, freeway
horse	pony, donkey, pig, goat, mule
computer	computers, pc, crt, chip, compute

Figure 25: [46] Some examples of the nearest neighbors learned by the model.

The authors from the architecture here presented scored first place in the

COCO 2015 image captioning challenge. They made slight modifications to their original model which were published in the follow-up paper [47]. The two most significant improvements were obtained by using a newer version of GoogLeNet (GoogLeNetv3) and reducing the beam search size from 20 to 3.

The two metrics used in the challenge were: (M1) the percentage of captions that are considered better or equal as the reference human captions, and the second criteria (M2) is the percentage of computer generated captions that passed the Turing Test. They obtained a score of .273 and .317 for M1 and M2 respectively, whereas human captions obtained the ratios of .638 for M1 and .675 for M2. Finally they also reported highest METEOR score with a value of .254.

5.6 Additional models for image captioning

5.6.1 From captions to visual concepts and back

One of the first papers that proposed a CNN as an image encoder was the model proposed by Fang [15]. This approach proposes several of the most important ideas to construct what it is now the state-of-the-art model for image captioning. Their approach consisted on first identifying the 1000 most frequent words in the COCO training dataset. Then they trained a model to extract words from the image. This word extraction model consisted on modifying a pre-trained CNN; either AlexNet or VGG16, by converting the fully connected layers into convolutional layers. They located on top of it a modified version of Viola-Jones cascade detectors in order to output a probability for all 1000 words in every image. Once they trained a model to acquire a set of words from every image, they proceeded by training a language model that takes as input words and constructs a sentence. The proposed language model is a maximum entropy (ML) language model, which estimates the probability of a word given a previous set of words in the sentence. After generating a set of sentences by using a beam-search on the language model, they proceeded to re-rank the sentences using another pre-trained model that they called deep multimodal similarity model (DMSM). The main idea behind this model is to learn two neural network embeddings such that one neural network maps images to a vector space, and the other neural network learns to map sentences into to a vector space of the same dimension. Once images and sentences are embedded on the same space they measure their resemblance using cosine similarity. An overview of the general pipeline can be observed in image 26. One disadvantage of this architecture is that it consists of several models that have to be trained separately and then merged.

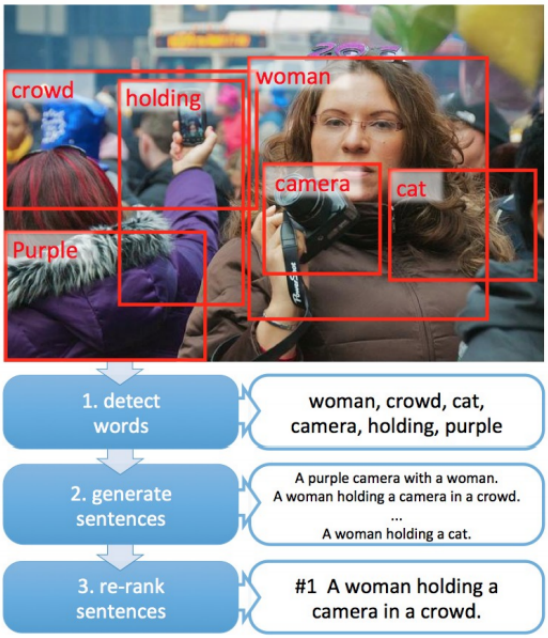


Figure 26: [15] General pipeline used by Fang [15].

5.6.2 Show, attend and tell

Xu [48] presented a modified version of [46] that incorporates visual attention. Visual attention lets the model focus on certain aspects of the image at every time step. Therefore the attention module provides a clear visualization of what the model is focusing on. This single aspect could help us understand the errors made by the model. An overview of the model is displayed in figure 27.

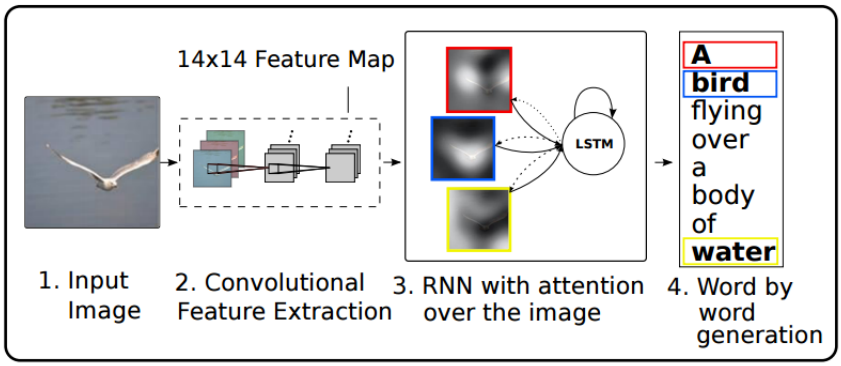


Figure 27: [48] This model is very similar to NIC presented in [46]; however, with the inclusion of the attention mechanism it learns to focus at different locations at every time-step.

This model extracts image features from a pre-trained CNN (VGG16); however, instead of using one of the last layers of a CNN like in [47], they extract the feature maps from the initial layers. The attention modules take as input these feature maps along with the previous hidden state of the LSTM and transform it into a single feature map for which the sum of all its values equals one. This transformation is done using a dense layer using a softmax activation function. This single feature map is what we refer to as the attention feature map. We can use the attention feature map to obtain the relevant sections of the image by multiplying every feature map from the original extracted features with the attention feature map. We will refer to the result of this multiplication as the feature map context since it incorporates a weighted information of the image. Since all of these operations are differentiable we can use any gradient descent method for training; moreover, the model was trained end-to-end using Titan Black GPU and it took less than 72 hours [48]. The METEOR score calculated for the COCO dataset using the same data splits from [28] was of 23.90. The attention model lets us visualize where was the model focusing on in the image when it displays as incorrect word, this behaviour is observed in figure 28. A compete visualization for every word in a sentence can be observed in figure 29.

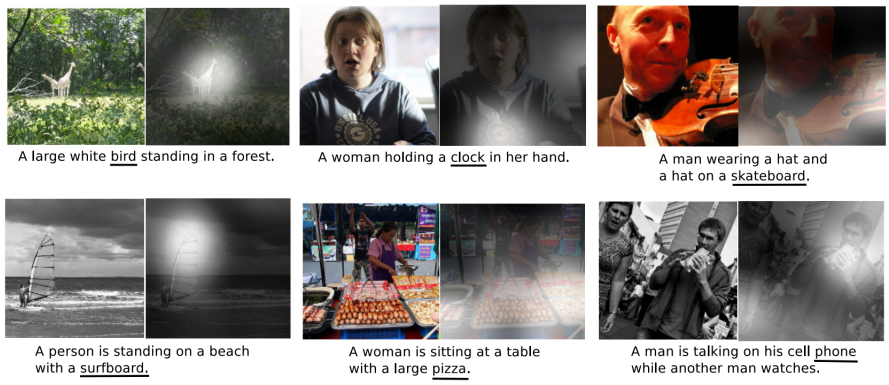


Figure 28: [48] Using the attention context we can observe at which part of the image the model is focusing on when it outputs an incorrect word. The attention of the underlined word is displayed with the white blobs in the image.

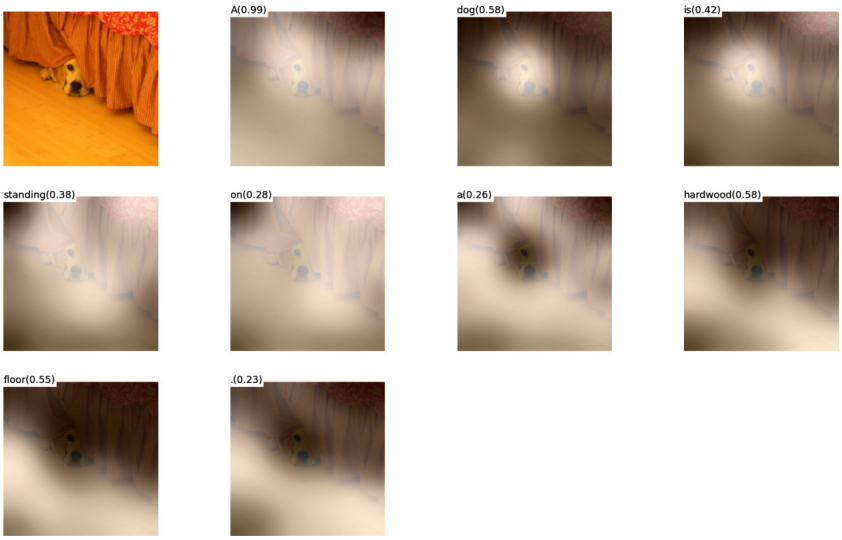


Figure 29: [48] The attention model at every time step for the sentence "A dog is standing on a hardwood floor". An important remark is how the attention learns to focus on the dog when the model outputs the word "dog", and also how it learns to take the complement of the dog when it outputs the second "a" which carries is calculated by also taking into consideration the hidden state crated by the previous sequence "A dog is standing on"
.

6 Image segmentation

Modern advances in computer vision tasks include image classification and accurate bounding box detection. A natural step towards a more challenging task is image segmentation. Image segmentation makes classification at pixel-level; therefore, provides a finer localization of the objects than the ones presented by bounding boxes. In this section we proceed to review the current state-of-the-art architecture for image segmentation.

6.1 Fully convolutional networks

Similarly to image captioning, image segmentation uses a pre-trained CNN as an image encoder. The current state-of-the-art model [33] used VGG16 to encode the image and an upsampling technique called transposed convolution to decode the image into its original shape. The decoded image has the same width and height of the original image; however, it contains N feature maps that correspond to the N classes plus background. Thus, every feature map encodes the probability that a certain class or background is located in the image. Transposed convolutions were introduced used in [51] as a visualization technique in order to project the feature maps into the original image dimension.

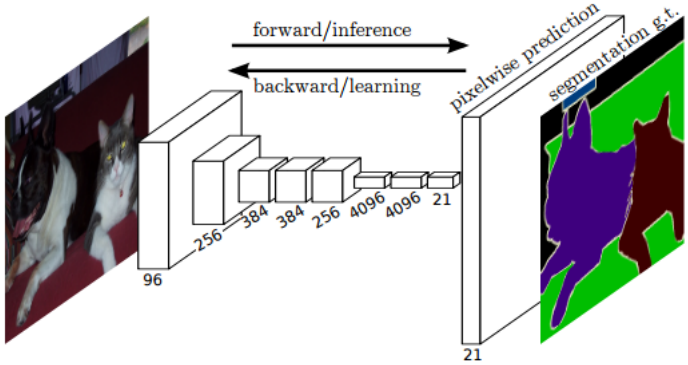


Figure 30: [33] Fully convolutional network proposed for image segmentation. The encoding section corresponds to a modified VGG16 that changes the fully connected layers for convolutional layers.

Another important aspect of this architecture is that it does not contain any fully connected layers; consequently, it's referred to as a fully-convolutional network (FCN). We can observe an FCN network on figure 30.

Contrary to normal CNNs, FCNs can operate using as input any image size by slicing the network across the image [33]. After convolving, they

apply the transposed convolution operator to return to the original image dimensions. In order to convert the fully connected layers into convolutional layers, the authors undo the flattening operations by rearranging the dense layers back into kernels. Then they use in the last convolution, a convolution of 1000 feature maps with a kernel size of 1×1. The authors from [33] refer to this operation as *convolutionalization*, this operation can be visualized in figure 31.

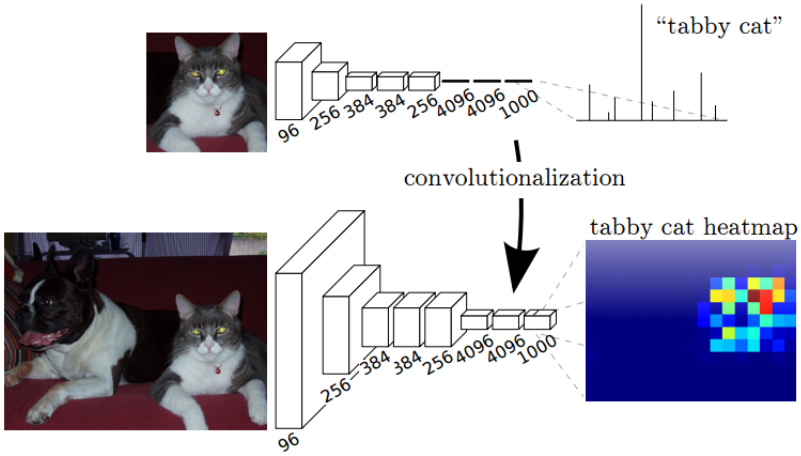


Figure 31: [33] Transformation from a CNN into FCN by rearranging the dense layer into kernels. As seen in the top image, classification takes a pre-defined section of the image and classifies it; meanwhile, a FCN network performs a convolution around the complete image and is able to display heatmap for every class.

After passing the image through the FCN and obtaining a heatmap, the authors perform a transposed convolution. A single transposed convolution is performed in their model. The transposed convolution operation can be visualized in figure 32. As mentioned in [33] a transposed convolution upsamples the image by a factor f, by incorporating strides between the values of the heatmaps, this can be visualized in figure 33.

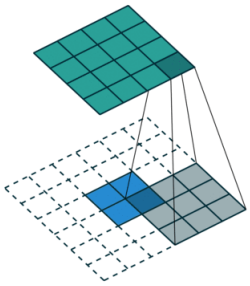


Figure 32: [13] Transposed convolution with zero stride and zero padding. The square in blue represents the previous feature map, the squares in gray represent the kernels and the green squares the outputted feature map. Consequently, the gray squares move sequentially around the zero padded feature map performing a convolutional operation. We can observe that the feature map yielded by the transposed convolution is bigger than the original feature map.

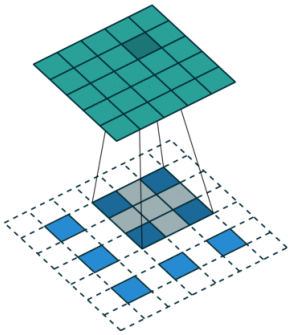


Figure 33: [13] Transposed convolution with zero padding and a stride of two.

Another important aspect of the network is that the authors applied a transposed convolution not only on the last feature map, but along distinct feature maps on the architecture. The authors up-sampled the feature maps at three distinct parts of the network; specifically in the pool3 and pool4 layers of the VGG16 network as seen in figure 34. They call this technique *deep-jet*. The results of using the *deep-jet* method are visualized in figure 35.

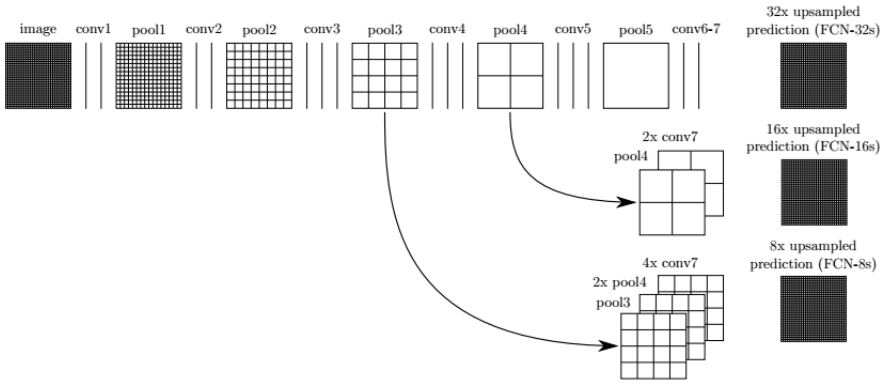


Figure 34: [33] This image presents the three possible outputs from the proposed deep-jet model. These outputs are presented at the right of the image with the names FCN-32s, FCN-16 and FCN-8s. FCN-32s only uses the last convolutional layer to upsample using a 32 stride transposed convolution. FCN-16s combines both the feature maps from pool4 and the one from the last convolutional layer by upsampling two transposed convolutions from the last feature maps and summing it up with the feature map of pool4 passed through a convolution with a kernel size 1×1. Finally this feature maps are upsampled to the original image size. They perform a similar operation with the FCN-8s which combines the feature maps from the last layer, pool4 and pool3.

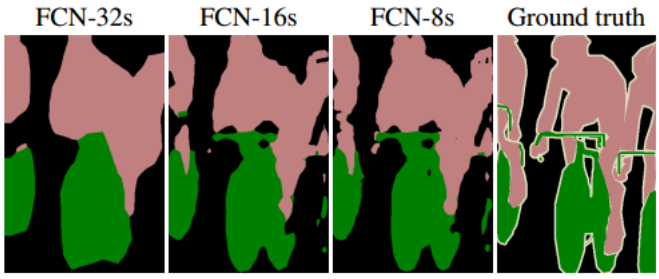


Figure 35: [33] This image shows how the segmentation gets finer by upsampling at different locations of the FCN. As described in figure 34 FCN-32s only uses the last feature maps, and FCN-16 and FCN-8s incorporates information from intermediate sections of the FCN.

This architecture was trained using SGD with momentum. The batch size was set up to 20 and a fixed learning rate of 10^{-4} while using VGG16. Momentum was set to .9. They obtained state-of-the-art results on PASCAL VOC 2011 and 2012 datasets, scoring a mean intersection over union (IU) of 62.7 on VOC2011 and a mean IU of 62.2 on VOC12. This results are 20% better that the previously scored results [33] which obtained 52.6 and 51.6 IU in VOC2011 and VOC2012 respectively.

7 Image captioning for robotics

In this section we discuss in detail our implementation of the image captioning model for the purpose of robot platforms and anomaly detection. First, we will present a technical comparison between the most common CNN architectures used in image captioning and scene segmentation. We then proceed by summarizing all details of our image captioning implementation. Next, we discuss the details used to create our 2016 anomaly detection dataset (ADD 2016), as well as our final architecture used for classification and captioning of these anomalies. We conclude by examining our results and proposing several modifications to the original image captioning architecture.

7.1 CNN benchmarks

Currently, there are four major open source frameworks for deep learning: Caffe, Tensorflow, Theano and Torch. All these frameworks support GPU programming trough NVIDIA's CUDA platform as well as NVIDIA's deep neural networks library (cuDNN). Also, all of these except for torch are developed or have a wrapper in python. Torch works on top of the programming language Lua. One obvious disadvantage related to the amount

of frameworks in the field, is that the current research community is divided between them. Consequently, improvements in one area such as image captioning are initially programmed in Tensorflow while state-of-the-art results for scene segmentation were obtained in Caffe. While there are tools for transforming pre-trained models between frameworks, it is still complicated to replicate and improve implementations between them.

In previous sections we discussed the current state-of-the-art implementations of image captioning and scene segmentation. Both of these methods used a pre-trained CNN in order to extract a vector of features from raw input images. Since our final goal is to implement a scene-understanding algorithm in a robot platform, we proceeded to further investigate the time it takes for different CNN architectures to process an input. All of the results from table 1 to table 5 were obtained by Johnson in [26] using CNNs with different GPUs under the Torch framework. The GPUs in which the tests were conducted can be seen in table 1.

GPU	Memory	Architecture	CUDA Cores	Release Date
Pascal Titan X	12GB	Pascal	3584	August 2016
GTX 1080	8GB	Pascal	2560	May 2016
Maxwell Titan X	12GB	Maxwell	3072	March 2015

Table 1: GPUs used to evaluate the CNNS.

We now display the results for the following architectures: AlexNet [30], VGG16[40] and GoogLeNet [44]. All of these results were calculated using cuDNN version 5.1.05 on a machine with 64GB RAM running Ubuntu 14.04. Also, the results were calculated for a minibatch of 16 images of sizes 224×224 [26]. The experiments were performed using 10 trials. An overview of all networks is displayed on table 2, and the specific details of AlexNet, GoogLeNet v1 and VGG16 results are presented in tables 3, 4 and 5 respectively.

Network	Layers	Top-5 error	Total speed (ms)
AlexNet	8	19.80	14.56
GoogLeNet V1	22	10.07	39.14
VGG16	16	8.80	128.62

Table 2: CNN overview. Total speed represents the sum of the backward pass and forward pass of a single minibatch using as GPU a Titan X pascal with cuDNN 5.1.

GPU	Forward (ms)	Backward (ms)	Total (ms)
Pascal Titan X	5.04	9.52	14.56
GTX 1080	7.00	13.74	20.74
Maxwell Titan X	7.09	14.76	21.85

Table 3: AlexNet evaluation.

GPU	Forward (ms)	Backward (ms)	Total (ms)
Pascal Titan X	12.06	27.08	39.14
GTX 1080	16.08	40.08	56.16
Maxwell Titan X	19.29	42.69	61.98

Table 4: GoogLeNet V1 evaluation.

GPU	Forward (ms)	Backward (ms)	Total (ms)
Pascal Titan X	41.59	87.03	128.62
GTX 1080	59.37	123.42	182.79
Maxwell Titan X	62.30	130.48	192.78

Table 5: VGG16 evaluation.

7.2 Captioning anomalies

Anomaly detection is one possible application of image captioning in robotics. A robot could classify anomalies such as broken windows, injured people or car accidents. However, instead of making a simple classification between having an anomaly or not, the robot would now be able to communicate with another human being; using a complete natural sentence, the anomaly that it perceives. A complete sentence brings more information about the situation than the classes detected on the image. For example, an image that was classified as an anomaly but only reports the class "people" brings little information about the subject to attend; however, a system that classifies the situation as an anomaly but also communicates "Five people are fighting." could prove more beneficial in order to address such situation. A robot with this ability could be deployed in hospitals, houses, supermarkets as either a mobile or a static platform.

7.2.1 Results of the anomaly detection dataset

We created a dataset that contains 1008 captioned images which only correspond to anomalies (AD dataset). We consider that an image is an anomaly if it contains one of the following classes: broken windows, injured people, fights, car accidents, fire accidents , guns or domestic violence. These classes were specifically selected in order to develop robot activities such as: patrolling or domestic-services. To the best of this author's knowledge, there exist a limited amount of datasets that contain these anomalies. However, those that exist suffer from some deficiencies regarding our application, mainly: none of them are captioned, they only provide a limited amount of the desired classes and some of them are not easily accessible. Specifically, the authors from [36] presented a dataset that only contain fights between hockey players, and the authors from [11] created a dataset which is made on violent scenes from movies; however, this dataset can't be easily distributed due to copyright infringement. All images from our dataset were selected from flickr using all creative commons license which allow us distribute the

dataset under certain conditions such as: attributing the original authors and a non-commercial agreement. All the images were captioned by 20 different persons, who were asked to follow the next set of instructions in order to caption an image:

- Write a single English sentence for each image.
- The sentences have to be in present or present continuous tense. Present continuous tense: Present tense of the verb "to be" plus the present participle (-ing form) of a verb i.e. "a man is laying on the ground, while two paramedics are assisting him".
- Write primarily about the accident/incident/anomaly in the image.
- When possible be explicit with the number of persons in the image.
- The sentence should not contain any digits (i.e. 1 2 3 ... 9).
- Use written numbers instead of digits (i.e. 'one' 'two' ... 'nine').
- When appropriate be explicit with the gender (man, woman).
- There is no limitation in the length of the sentence. However, it is advised to use between 7 to 18 words per sentence.

Some examples of the images presented in our dataset can be seen in figure 36.

Figure 36: Images from our anomaly detection dataset. The captions used for these images are: a) two policemen carry rifles on the street, while one of them is aiming at something b) a woman with an injured leg is sitting on the side of the road next to her bicycle c) two men kicking violently the windshield of a car d) there is something burning in front of a shop.

7.3 Technical details of our image captioning model

In order to validate our model we trained on the COCO training dataset and the IAPR 2012 dataset [22]. The IAPR 2012 dataset consists of 20,000 captioned images. Two of the main characteristics of this dataset are: first it is free of any copyright restrictions; therefore it can be distributed easily, and second, the dataset is very diverse. For example, the dataset contains images that describe human activities such as pushing, drinking, celebrating, but also contains examples of wildlife, city pictures and landscapes. This dataset has also being used by [14] in order to validate their implementation and create a multi-lingual image captioning model. An important consideration that we would like to pinpoint in both dataset is the bias they could have in regards to certain specific words. Specifically we found that in the COCO dataset the word "man" is the ninth most repeated word with 51222 instances; however, the word "woman" is used less than half with respect to the word "man" and holds the position nineteenth with 23994 instances. These sort of unbalances can cause the model to predict a word more than another in a real world scenario, in this specific case to predict more the word "man" instead of "woman" when it detects any person.

We preprocessed our data similarly to the image captioning models described in previous sections [46] [28]. Therefore, the pre-processing step consisted on removing all non-alphanumeric symbols, converting all sentences to lowercase and removing all words which had a frequency less than 3 for both datasets. We then created a one-hot encoding representation of every word. Therefore the dimension of the one-hot encoding represents our vocabulary size. The vocabulary size for COCO was of 9413 words, while in IAPR was of 1078. One main difference in our preprocessing step is that we discarded all captions that were longer than 16 for COCO and 14 for IAPR. This was done since we are only interested in developing sentences that are able to describe anomalous situations in a concise manner.

After preprocessing all the target sentences we proceeded to process all the images. This step consisted of passing all images through a beheaded CNN. For this purpose we used as CNN Inception-V3, which corresponds to the third incarnation of GoogLeNet network. Since this network was not tested in the benchmarks provided by [26], we decided to provide a test of 30 forward passes using 4 popular CNNs available in Tensorflow. These results are displayed in figure 37.

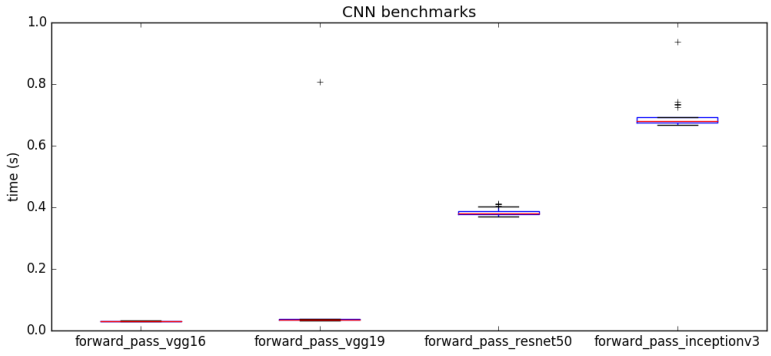


Figure 37: Results for 30 forward passes using VGG16, VGG19, Resnet50 and InceptionV3. All models were tested using a single image as a mini-batch, a GTX1070, Tensorflow, and CUDA 8.0 with cuDNN v5.1.

This particular trained Inception V3 network obtains top-5 error of 7.3 % on ImageNet [7]. The current implementations of image captioning models use a beam search in order to refine their sentences. This step could prove computationally expensive for any real-time implementation; therefore, we chose to use Inception-V3 since it has the lowest top-5 error on ImageNet when compared to the other pre-trained networks found in our deep learning framework [7]. Passing all the COCO images trough the beheaded Inception-V3 took approximately 1.5 hours using the same computational set up described in figure 37. The original input to the CNN (299×299 pixels) got encoded in a feature vector of 2048 dimensions. Every feature vector was saved in a hdf5 dictionary. The COCO training dataset contains 415715 captions from 82783 images [32]. This corresponds on 13 GB of data. Loading all these images in memory proves infeasible; however, we only need to load a mini-batch since we are training using a variation of stochastic gradient descent. The appropriate programming paradigms in the python language that let us yield a section of our dataset per training step are called generators. Our final architecture used 512 LSTM neurons and we used as in the state-of-the-art a word and image embedding of 512 dimensions. We split both datasets in the same manner: 80 % of all images were reserved for training and validation, and the 20 % left as a test set. We created two generators one for disposing training data, and another for validation data. We trained the both COCO and IAPR datasets for 50 epochs using ADAM as our optimizer [29] Every epoch goes through all captions. Going through the 50 epochs took approximately 9.5 hours for the COCO dataset and 4.5 minutes for IAPR data.

8 Results

In order to validate our pre-trained models we use the metric METEOR implemented in [8]. However, one important remark that we would like to emphasize; and which has not been thoroughly discussed in any of the papers here reviewed, is the correlation between the loss function and METEOR. We have found empirically, that in all of our tested trained models, we obtained better results not when the validation loss or accuracy was at its minimum, but several epochs after it. We argue that the existence of this discrepancy between the loss function and the METEOR score, is due to the way in which the target variables are represented in all image captioning models. The target values are one-hot encodings; consequently, they locate all probability mass in a single word. This could also be considered as representing our target values as a hard-evidence [3]. However, the METEOR metric uses also a synonym matching module. Therefore, there is an underlying symmetry between words, which makes the METEOR metric invariant under specific switching between the one-hot encoding representations. This leads us to think that we can relax the hard-evidence constraint and use instead a soft representation of the targets. This could be done by distributing uniformly the probability of every single word along its synonyms, or by considering any pre-trained word embedding model and distributing the probability using a Gaussian distribution centered on the original word. Testing these hypothesis goes beyond the scope of this research and we leave this for future work.

8.0.1 COCO results

After 50 epochs the METEOR score obtained for the COCO dataset is 14.2; this score is 7 points below the-state-of-the-art. However, we are not performing any beam-search before predicting our captions. We display some of the results in figures 38 and 39.

(a) a man is doing a trick on a skateboard

(b) a train traveling down train tracks next to a lush green field

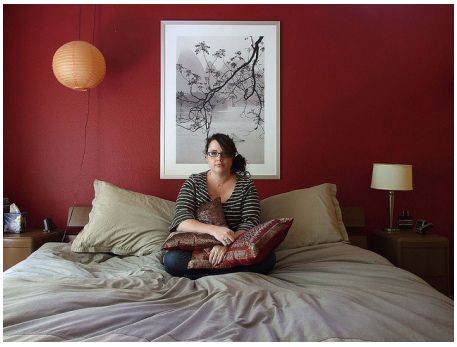
(c) a group of people flying kites in a field

(d) a pizza with cheese and herbs on a table

(e) a desk with a laptop and a monitor

(f) a man on a surfboard riding a wave in the ocean

Figure 38: A selection of correct captions performed by our model in the COCO dataset.



(a) a woman sitting on a bed with a laptop computer

(b) a dirty toilet in a small room with a wooden floor

(c) a cat sitting on a car seat with a concerned look

(d) a cup of coffee and a cup of coffee

(e) a man is eating a sandwich and drinking beer

(f) a man and woman are holding wine glasses

Figure 39: A selection of incorrect captions performed by our model in the COCO dataset.

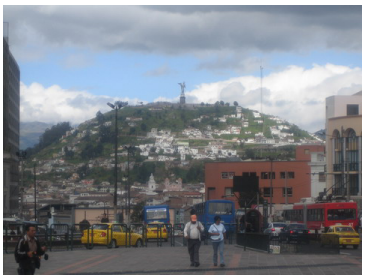
We can observe in figure 38 some of the correct captions displayed by our model. Sub-figure (a) of the same figure 38, shows that our model is able to localize small objects such as a "skateboard". Another important remark is the presence of non-iconic images. For example in sub-figures (a), (c) and (e) there is not a salient object that should be described; instead, the objects in the figures are often cluttered or hardly visible due to the background. Sub-figure (c) indicates that the model is able to recognize and describe objects that occur several times in the images, e.g. "a group of people" and "kites". However, we will later comment on how our current model is unable to count. Sub-figure (e) describes correctly a cluttered environment, indicating two of the most significant objects in the scene such as "laptop" and "monitor". Sub-figures (b), (d) and (f) display iconic images which are correctly captioned.

We can observe in figure 39 several errors made by our model when trained on the COCO dataset. Typical errors include misclassification of objects and scenes along with the inability to count. For example in (a) of the same figure, a woman is sitting on a bed; however, it does not have a "laptop" but a pillow. Sub-figure (b) is described as a "small room"; however it is clearly located outside. It also describes "a wooden floor" which could be a reference to the wooden fence located behind the toilet. Sub-figure (c) located the cat in a "car seat". This sub-figure is a misclassification of the scene. Another interesting point regarding this image is the word "concerned". Looking further on the training dataset we found that there were multiple cases in which the captions described animals using this word. Sub-figure (d) shows an interesting behaviour of our model, which is the inability to count. This behavior is repeated several times in different results (sub-figure (f) of the same image). Even tough the caption is semantically correct, it is not grammatically optimal. Sub-figure (e) displays two misclassification "sandwich" and "beer". Sub-figure (f) also displays two misclassifications "woman", "wine glasses" along with the inability to count the number of persons in the image. Overall our model is able to recognize the general aspects of the scenes: Sub-figure (a) "woman sitting on a bed", sub-figure (b) "a dirty toilet", (c) "cat", "car" (d) "a cup of coffee", (e) "a man eating" and (f) "a man ... holding".

Finally we would like to mention that all our trained models didn't have any prior knowledge of the language or grammar. And putting aside the description of the image, it learned how to construct complete English sentences.

8.0.2 IAPR results

We obtained a METEOR score of 16.6 in the IAPR dataset. This score is better than the reported best for this dataset [14] (15.4). However we emphasize that we only used captions that were at most 14 words long. Figure 40 displayed below shows a selection of these results.



(a) a city with cars and people in the foreground

(b) portrait of a black girl with black hair and a white tee-shirt

(c) two tennis players on a green hard court

(d) a paved road in the middle of a flat dry desert

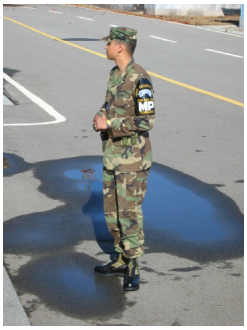
(e) a sea lion at a sandy beach

(f) fireworks in the harbour of a city at night

Figure 40: A selection of correct captions performed by our model trained on the IAPR dataset

(a) three men are sitting in front of a white bus on a wooden

(b) view of a river with a river in the middle of the jungle



(c) two men are standing in front of a grey van

(d) a pink tower with three blue windows

(e) people on a somewhat red square with many trees

(f) three men are playing volleyball on a brown sandy beach in the

Figure 41: A selection of incorrect captions performed by our model trained on the IAPR dataset

Figure 40 displays correct captions performed by our mode when trained on the IAPR dataset. Similarly to the COCO dataset, our model learned to caption images which are cluttered. For example sub-figure (a) of the same figure 40, displays a very complicated scene surrounded by cars and houses. However, it was able to describe the most import objects along with a grammatically correct sentence. Sub-figure (b) shows the ability of the model to distinguish between colors located in the same image "black girl", "black hair", "white tee-shirt". Sub-figure (c) also displays the ability to distinguish colors "green". Sub-figure (d) shows an interesting result since it displays how our model described an image which consists of objects only located as a part of the background: "paved road" and "dry desert". This shows us that the model is able to describe not only salient objects but also general scenes. Sub-figure (e) show us an iconic-image of a sea lion along with the background. Sub-figure (f) displays a hardly visible scene. However, the generated caption is able to make references to objects in the image such as: "city", "harbour", "fireworks", and also make a temporal reference "at night".

Figure 41 displays incorrect captions performed by our model trained on the IAPR dataset. One important remark that we observed in these captions is that even when the model is trained on different datasets, it still displays similar errors. Mostly, misclassification of objects and the inability to count. However, by training on a smaller dataset (IAPR) we obtained captions that are not always complete. For example, we can observe in sub-figure (a) of the same figure 41, that the sentence ends without describing the final object "on a wooden". It also misclassifies the object by saying "in front of a white bus". One hypothesis is that the model believes that the "white bus" is composed of the white background along with the windows displayed at the end. Sub-figure (b) show a similar behavior displayed in model trained in COCO. Specifically, it repeats object names; however, in this particular case there is only a single river. Sub-figure (c) shows a misclassification regarding "a grey van". It also counts incorrectly the number of persons located in the image. Sub-figure (d) displays an interesting result regarding colors. Two questions that we leave for further research are: when does the model starts changing the classification of the colors? And whether or not we can embedded a more diverse color palette? One possible test could be done by training along input data that consists of an image of a single color, and a caption of one or two words which describes the color name. Sub-figure (e) supports the hypothesis that the color palette of the model should be developed further. Finally sub-figure (f) misclassified the activity "playing volleyball"; however, it indicated correctly the foreground "on a brown sandy beach". Similarly to sub-figure (a) it ended the sentence incorrectly.

8.0.3 IAPR-ADD results

After 50 epochs our model trained on the IAPR and AD datasets obtains a METEOR score of 16.2. In figure 42 we display only the results from the images that correspond to the anomaly detection dataset.

(a) a man is holding a gun

(b) a house is burning

(c) a car is crashed in a snow covered street

(d) there is a woman with blood on the floor

(e) a man is choking another man

(f) a firefighter is trying to put out a fire

Figure 42: A selection of correct captions generated by our model when trained on the IAPR dataset and the anomaly detection dataset.

(a) people with red helmets are sitting and cars on a

(b) there is a broken window laying on the ground

(c) a man is showing his injured shoulder

(d) a man showing his right arm in which he has severe injury

(e) a man is being held by the police

(f) a man is dancing with a woman on the dancefloor

Figure 43: A selection of incorrect captions generated by our model when trained on the IAPR dataset and the anomaly detection dataset.

Figure 42 shows correct captions performed by our model when trained on the IAPR and the AD dataset. The images here selected, display possible robotic applications which consists of detecting and describing accidents, fires and car crashes. In the sub-figure (a) of the same figure 42, the model correctly describes the anomaly by referring to a "gun". Sub-figure (b) correctly displays the situation of a house burning. We can also notice how concise the descriptions are, since it described the image correctly by using only 4 words. Sub-figure (c) made a reference to a car accident "a car is crashed" but also to the background "snow covered street". Sub-figure (d) also displays in a very concise manner a possible accident by making reference to "woman" "blood" "floor". Sub-figure (e) shows an example of how our model can be superior to a classification model in regards of anomaly detection. Using a simple classification algorithm we could have only detected "man". However, our model not only detects two men but also detects the their activity by stating "is choking". These two words bring more information about image which could prove beneficial when addressing such situation. Sub-figure (f) is an important results since it was able to communicate the existence of a possible fire but also the contribution of a "firefighter" in the scene "trying to put out a fire".

Figure 43 displays the incorrect captions performed in the IAPR and AD dataset. Since the AD dataset was trained along the IAPR dataset, it suffers from the same problems. Mostly, misclassifications. Sub-figure (a) from the same image 43 displays an incorrect classification of "helmets" and "cars". It also finished the sentence abruptly. Sub-figure (b) misclassified the word "bike".

Sub-figure (d) shows a complicated image in which the leg of man is misclassified as an "arm". This sort of classifications are difficult to any CNN since they are not intrinsically invariant to rotations. Sub-figure (e) misclassified the referees by "police" officers. And finally sub-figure (f) misclassified the scene and the activity using "dancing" and "dancefloor".

We added to our existing CNN model a multi-layer perceptron with two hidden layers in order to provide a binary classification between normal situations and anomalies. We categorize as positive an image that is not an anomaly, and as negative an image that corresponds to an anomaly. We report an accuracy of 97 %. The confusion matrix with the number of images is displayed in table 6 and the confusion matrix with ratios is displayed in table 7.

		Prediction label		
		Positive	Negative	Total
True label	Positive	TP = 1170	FN = 20	1190
	Negative	FP = 12	TN = 117	129
	Total	1182	137	1319

Table 6: Confusion matrix of the CNN model trained with the IAPR-AD

dataset.

		Prediction label		
		Positive	Negative	Total
True label	Positive	TP = .887	FN = .015	.902
	Negative	FP = .009	TN = .088	0.097
	Total	.896	.103	1

Table 7: Confusion matrix of ratios for our CNN model trained with the IAPR-AD dataset.

9 Conclusions

In this work we reviewed the state-of-the-art models for image captioning and scene segmentation. Both of these architectures used specialized CNNs; consequently, in order to create a detailed explanation of the models, we provided a theoretical review of CNNs as well as LSTMs. The theoretical background explains the forward pass and backward pass of ANNs, CNNs and LSTMs. We proceeded by reviewing GoogLeNet, the VGG networks, the METEOR metric, the COCO Microsoft dataset, as well as recent variations of the image captioning model. Next, we investigated relevant technical details behind the most used CNNs in scene-understanding algorithms. Some of these included the top-5 error on ImageNet, the number of layers, as well as the forward-pass and backward-pass speeds using different GPUs. We then created an image-captioning model that considers robot-platforms for anomaly detection. In order to validate our model we trained it on the COCO and IAPR datasets, obtaining significant results on both datasets. Finally, we created an anomaly detection dataset consisting of 1008 images and captions, which we used along the IAPR dataset in order to create an architecture that predicts and communicates anomalies in natural language. We obtained an accuracy of 97 % and a METEOR score of 16.2.

References

[1] Executive summary: World robotics 2016 service robots. *International Federation of Robotics*, 2016.

[2] Satanjeev Banerjee and Alon Lavie. Meteor: An automatic metric for mt evaluation with improved correlation with human judgments. In *Proceedings of the acl workshop on intrinsic and extrinsic evaluation measures for machine translation and/or summarization*, volume 29, pages 65–72, 2005.

[3] David Barber. *Bayesian reasoning and machine learning*. Cambridge University Press, 2012.

[4] Ian Goodfellow Yoshua Bengio and Aaron Courville. Deep learning. Book in preparation for MIT Press, 2016.

[5] A Borsellino and T Poggio. Convolution and correlation algebras. *Kybernetik*, 13(2):113–122, 1973.

[6] Xinlei Chen, Hao Fang, Tsung-Yi Lin, Ramakrishna Vedantam, Saurabh Gupta, Piotr Dollár, and C. Lawrence Zitnick. Microsoft COCO captions: Data collection and evaluation server. *CoRR*, abs/1504.00325, 2015.

[7] Franois Chollet. keras. https://github.com/fchollet/keras, 2015.

[8] Jonathan H Clark, Chris Dyer, Alon Lavie, and Noah A Smith. Better hypothesis testing for statistical machine translation: Controlling for optimizer instability. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies: short papers-Volume 2*, pages 176–181. Association for Computational Linguistics, 2011.

[9] Adam Coates, Paul Baumstarck, Quoc Le, and Andrew Y Ng. Scalable learning for object detection with gpu hardware. In *2009 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 4287–4293. IEEE, 2009.

[10] Matthieu Cord. Heuritech: Advanced semantic analysis. Deep CNN and Weak Supervision Learning for visual recognition https://blog.heuritech.com/2016/02/29/a-brief-report-of-the-heuritech-deep-learning-meetup-5/, 2016. Accessed: 24-12-2016.

[11] Claire-Hélène Demarty, Cédric Penet, Mohammad Soleymani, and Guillaume Gravier. Vsd, a public dataset for the detection of violent scenes in movies: design, annotation, analysis and evaluation. *Multimedia Tools and Applications*, 74(17):7379–7404, 2015.

[12] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pages 248–255. IEEE, 2009.

[13] Vincent Dumoulin and Francesco Visin. A guide to convolution arithmetic for deep learning. *arXiv preprint arXiv:1603.07285*, 2016.

[14] Desmond Elliott, Stella Frank, and Eva Hasler. Multi-language image description with neural sequence models. *CoRR*, abs/1510.04709, 2015.

[15] Hao Fang, Saurabh Gupta, Forrest Iandola, Rupesh K. Srivastava, Li Deng, Piotr Dollar, Jianfeng Gao, Xiaodong He, Margaret Mitchell, John C. Platt, C. Lawrence Zitnick, and Geoffrey Zweig. From captions to visual concepts and back. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2015.

[16] Justin Johnson Fei-Fei Li, Andrej Karpathy. Convolutional Neural Networks for Visual Recognition. http://cs231n.github.io/. Accessed: 10-12-2016.

[17] Felix A Gers and Nicol N Schraudolph. Learning Precise Timing with LSTM Recurrent Networks. 3:115–143, 2002.

[18] Ross Girshick. Fast r-cnn. In *The IEEE International Conference on Computer Vision (ICCV)*, December 2015.

[19] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. *Proceedings of the 13th International Conference on Artificial Intelligence and Statistics (AISTATS)*, 9:249–256, 2010.

[20] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In *Aistats*, volume 15, page 275, 2011.

[21] Alex Graves. Neural networks. In *Supervised Sequence Labelling with Recurrent Neural Networks*, pages 15–35. Springer, 2012.

[22] Michael Grubinger, Paul Clough, Henning Müller, and Thomas Deselaers. The iapr tc-12 benchmark: A new evaluation resource for visual information systems. In *International Workshop OntoImage*, volume 5, page 10, 2006.

[23] Grzegorz Gwardys. Convolutional neural networks backpropagation: from intuition to derivation. Experience with Machine/Deep learning https://grzegorzgwardys.wordpress.com/2016/04/22/8/, 2016. Accessed: 19-12-2016.

[24] David Jacobs. Correlation and Convolutionclass notes. Class Notes for CMSC 426, fall 2005.

[25] Max Jaderberg, Karen Simonyan, Andrew Zisserman, and Koray Kavukcuoglu. Spatial transformer networks. *CoRR*, abs/1506.02025, 2015.

[26] Justin Johnson. Cnn benchmarks. Open repository for CNN benchmarking https://github.com/jcjohnson/cnn-benchmarks, 2016. Accessed: 24-12-2016.

[27] Justin Johnson, Andrej Karpathy, and Fei-Fei Li. Densecap: Fully convolutional localization networks for dense captioning. *CoRR*, abs/1511.07571, 2015.

[28] Andrej Karpathy and Li Fei-Fei. Deep visual-semantic alignments for generating image descriptions. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2015.

[29] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[30] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In F Pereira, C J C Burges, L Bottou, and K Q Weinberger, editors, *Advances In Neural Information Processing Systems*, pages 1–9. Curran Associates, Inc., 2012.

[31] Rémi Lebret, Pedro H. O. Pinheiro, and Ronan Collobert. Phrase-based image captioning. *CoRR*, abs/1502.03671, 2015.

[32] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. Microsoft coco: Common objects in context. In *European Conference on Computer Vision*, pages 740–755. Springer, 2014.

[33] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional networks for semantic segmentation. *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 07-12-June-2015:3431–3440, 2015.

[34] Thomas M. Mitchell. *Machine Learning*. McGraw-Hill, Inc., New York, NY, USA, 1 edition, 1997.

[35] Kevin P Murphy. *Machine learning: a probabilistic perspective*. MIT press, 2012.

[36] Enrique Bermejo Nievas, Oscar Deniz Suarez, Gloria Bueno García, and Rahul Sukthankar. Violence detection in video using computer vision techniques. In *International Conference on Computer Analysis of Images and Patterns*, pages 332–339. Springer, 2011.

[37] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting on association for computational linguistics*, pages 311–318. Association for Computational Linguistics, 2002.

[38] Bryan A Plummer, Liwei Wang, Chris M Cervantes, Juan C Caicedo, Julia Hockenmaier, and Svetlana Lazebnik. Flickr30k entities: Collecting region-to-phrase correspondences for richer image-to-sentence models. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2641–2649, 2015.

[39] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks. *ArXiv 2015*, pages 1–10, 2015.

[40] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[41] Nitish Srivastava, Geoffrey E Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[42] Russell Stuart and Norvig Peter. Artificial intelligence-a modern approach 3rd ed, 2016.

[43] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. pages 3104–3112, 2014.

[44] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 07-12-June-2015:1–9, 2015.

[45] Ramakrishna Vedantam, C Lawrence Zitnick, and Devi Parikh. Cider: Consensus-based image description evaluation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4566–4575, 2015.

[46] Oriol Vinyals, Alexander Toshev, Samy Bengio, and Dumitru Erhan. Show and tell: A neural image caption generator. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3156–3164, 2015.

[47] Oriol Vinyals, Alexander Toshev, Samy Bengio, and Dumitru Erhan. Show and tell: Lessons learned from the 2015 mscoco image captioning challenge. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2016.

[48] Kelvin Xu, Jimmy Ba, Ryan Kiros, Kyunghyun Cho, Aaron C. Courville, Ruslan Salakhutdinov, Richard S. Zemel, and Yoshua Bengio. Show, attend and tell: Neural image caption generation with visual attention. *CoRR*, abs/1502.03044, 2015.

[49] Lecun Y., Bengio Y., and Hinton G. Deep learning. *Nature*, 521(7553):436–444, 2015.

[50] Sergey Zagoruyko, Adam Lerer, Tsung-Yi Lin, Pedro O. Pinheiro, Sam Gross, Soumith Chintala, and Piotr Dollár. A MultiPath Network for Object Detection. *1604.02135V1*, (1), 2016.

[51] Matthew D. Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. *CoRR*, abs/1311.2901, 2013.